Consecutivity in the `isl` Polyhedral Scheduler

Sven Verdoolaege *Alexandre Isoard*

Report CW 709, November 10, 2017

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Consecutivity in the `isl` Polyhedral Scheduler

Sven Verdoolaege[*] *Alexandre Isoard*[†]

Report CW 709, November 10, 2017

Department of Computer Science, KU Leuven

Abstract

The `isl` schedule is based on the Pluto scheduler, which is a successful polyhedral scheduler that is used in one form or another in several research and production compilers. The core scheduler is focused on parallelism and temporal locality and does not directly target spatial locality. Such spatial locality is known to bring performance benefits and has been considered in various forms outside and inside polyhedral compilation. For example, the Pluto compiler has some support for spatial locality, but it is limited to a post-processing step involving only loop interchange. Consecutivity is a special case of spatial locality that aims for stride-1 accesses, which can be useful for constructing burst accesses and for vectorization. Stride-1 accesses have been targeted by an approach based on one-shot scheduling, but it is fairly approximative and not directly transferable to a Pluto-style scheduler.

This report describes an approach for consecutivity that is integrated in a Pluto-style polyhedral scheduler. Both intra-statement and inter-statement consecutivity is considered, taking into account multiple references per statement and the integration into a component based incremental scheduler.

[*]Polly Labs and KU Leuven
[†]Xilinx

Contents

1 Introduction

A program is said to exhibit locality if it reuses some data element stored in some form of cache before it gets evicted. A distinction is usually made between temporal locality, where the same element is reused, and spatial locality, where the reuse may be of some other element that is loaded into the cache together with the element that was accessed first, e.g., because they share a cache line. Improving spatial locality therefore usually brings performance benefits by increasing cache hit rate. Note that spatial locality typically requires specific treatment since accesses to neighboring elements are not captured by traditional dependence relations.

Consecutivity is a special case of spatial locality, where consecutive accesses to memory access consecutive elements. It provides additional advantages as it facilitates memory access vectorization and usually allows the hardware cache prefetcher to successfully predict the next memory access. On some architectures, this also allows consecutive accesses to be coalesced into a single burst request. One such architecture is the FPGA architecture, where the clock frequency of the logic is usually a fraction of the clock frequency of the external memory interface. To compensate for this difference in frequency, it is recommended that wider memory accesses of a multiple of the width of the memory interface (i.e., vectorized loads and stores) be performed. Additionally, consecutive memory accesses can be coalesced into a single burst request that will be handled by the memory controller in the most efficient way, usually guaranteeing close to one (widened) memory access per cycle, thus fully utilizing the available memory bandwidth.

Xilinx (2017, Chapter 6) notably recommends using memory ports as wide as 512 bits (e.g., vectors of 16 elements for 32 bits integer) and bursting memory transfers from off-chip global memory. Xilinx (2017, Appendix B) goes into more details and suggests storing the data into temporary buffers in on-chip memory (BlockRAM) so as to freely perform all the memory accesses of each array in a few bursts. Note that those two transformations consist in coalescing memory accesses, and, as such, require those memory accesses to be in the same direction (to/from global memory). That is, there is little to no advantage in getting good consecutivity by mixing loads and stores as they could not be bundled together.

Additionally, because spatial locality sometimes conflicts with scheduling constraints or simply because no consecutive memory cells are being accessed, it is not always possible to achieve consecutivity on all memory accesses. In such cases, abandoning consecutivity on some accesses (usually from the same array) can help achieving consecutivity on the remaining accesses.

This report describes an extension to the `isl` scheduler that targets a restricted form of consecutivity. In particular, a program is said to exhibit *array consecutivity* for a particular array A if consecutively executed statement instances accessing A access either the same or consecutive elements of A in memory. In general, it is difficult to control array consecutivity directly and the techniques described in this report will therefore try to achieve *reference consecutivity* instead. This reference consecutivity essentially corresponds to stride-1 accesses. Before delving into the details of the consecutivity support and how it can be used, the report therefore first describes some previously published techniques aiming at stride-1 accesses and/or more general spatial locality. After a detailed description of consecutivity schedule constraints, the report briefly describes partial rescheduling, which allows consecutivity schedule constraints to be taken into account locally, and some matrix operations that are used internally. The report concludes with a brief discussion.

Note that this report only focuses on consecutivity and, in particular, does not explain how to ensure that the consecutive accesses can also be executed in parallel, which would be an additional requirement for vectorization. If the innermost tilable band in the generated schedule happens to be fully parallel, then this will be the case. Otherwise, additional techniques may be required. See also Section 6.

Prototype implementations of the core consecutivity support in the `isl` scheduler and of their use in the `PPCG` polyhedral compiler are available in the commits tagged `consecutivity_CW_709` of `git://repo.or.cz/isl.git` and `git://repo.or.cz/ppcg.git`.

2 Related Work

This section describes some prior art. For consistency with the rest of the report, some terminology has been changed compared to the original sources. Various terms are used for the array dimension that gets mapped linearly to memory, including "major" (Bastoul and Feautrier 2003a), "fastest changing" (Kandemir, Ramanujam, and Choudhary 1999, see Section 2.3) and "fastest varying" (Kong et al. 2013, see Section 2.11). In this document, it will simply be called "innermost" (as in C). Note that this assumes that (in general) the accesses are expressed as multi-dimensional index expressions and that, in particular, the accesses have not been linearized. Otherwise, some form of delinearization (Grosser et al. 2015) should be applied first.

The linear part of an index expression will be denoted by F and it will be split into an outer part G and the innermost part H. The linear part of the schedule transformation will be represented by T and its inverse by Q. This linear part T is sometimes split in an outer part T_1 and an inner part T_2. During the (row-by-row) construction of T, T_0 represents the schedule computed so far.

2.1 Wolf and Lam (1991a)

For an array reference $\mathtt{A}[F\mathbf{i} + \mathbf{c}]$, Wolf and Lam (1991a) define the directions of *self-temporal reuse* to be those in

$$\ker F. \tag{1}$$

Let

$$F = \begin{bmatrix} G \\ H \end{bmatrix}, \tag{2}$$

with H the last row of F, then the directions of *self-spatial reuse* are those in

$$\ker G. \tag{3}$$

They also consider group-temporal reuse and group-spatial reuse between different references to the same array, but only for uniformly generated references (Gannon et al. 1988). Note that they perform loop nest transformations rather than per statement transformations. Since all the references in a group are transformed in the same way, the direct analog for affine-by-statement transformations of "groups" would correspond to references within the same statement.

Their objective is to place loops that are involved in reuse directions innermost. They partition the original loop iterators into those that appear in reuse directions and those that do not. For example, if $i_1 + i_2$ is a reuse direction, then both i_1 and i_2 belong to the first group. They then apply their SRP (skew, reversal, permutation) algorithm (Wolf and Lam 1991b) twice, once to the loop iterators not involved in reuse directions and once to the loop iterators that

are involved. Tiling is then performed on the innermost loops. Reorderings of the point loops of this tiling are not considered because they do not affect reuse. Note that for the purpose of consecutivity as targeted by this report, such reorderings do play an important role.

2.2 Anderson et al. (1995)

Anderson et al. (1995) apply data layout transformation to make make data accessed by processor contiguous in memory. In particular, they apply strip-mining and permutation to the array dimensions in order to match the distribution (block, cyclic or block-cyclic) by moving the strip-mined dimension that identifies the processor into the outermost position. Data layout transformations are not considered in this report.

2.3 Kandemir, Ramanujam, and Choudhary (1999)

Kandemir, Ramanujam, and Choudhary (1999) focus on self-spatial reuse because the cases where group-spatial reuse brings an additional reuse dimension over self-spatial reuse are rare (Wolfe 1996). The criterion they use is that the transformed innermost loop iterator should only appear in the innermost array index function and should moreover appear there alone and with coefficient one. They explain that this criterion is stronger than strictly needed for spatial locality. In fact, it is sufficient to guarantee reference consecutivity, as explained in Section 3.1.1 below.

The authors perform loop nest transformation (not affine-by-statement), meaning that a single transformation matrix is computed. They first consider "the" LHS array and determine entries in $Q = T^{-1}$ that guarantee

$$FQ = \begin{bmatrix} X & \mathbf{0} \\ \mathbf{0}^{\mathrm{t}} & 1 \end{bmatrix} \tag{4}$$

for every choice (through permutation of the array indices) of the innermost array index. In the examples, the constraints always result in a single entry of Q being forced to be equal to 0 or 1. Then, they consider the RHS arrays and try to find a layout (array index permutation) that satisfies the same condition by looking for an index expression that is identical to the index expression of the LHS array that was placed innermost. If there is no such index expression, then they try to find a layout with

$$FQ = \begin{bmatrix} X & \mathbf{0} & \mathbf{0} \\ \mathbf{0}^{\mathrm{t}} & 1 & 0 \\ \mathbf{0}^{\mathrm{t}} & 0 & 0 \end{bmatrix}. \tag{5}$$

That is, they try to achieve self-temporal reuse in the innermost array index expression and self-spatial reuse in the next index expression. The layout choices for the RHS arrays may also result in additional entries of Q getting fixed.

Kandemir, Ramanujam, and Choudhary (1997) simply discard inverse transformation matrices that violate data dependence or that are not of full rank. Kandemir, Ramanujam, and Choudhary (1999) use the method of Li (1993) with "appropriate modifications" to complete the inverse transformation matrix to a full non-singular matrix that respects the dependences.

For multiple loot nests, they start with "most costly nest" and then continue with the other nests taking into account the now fixed choices for the innermost index expressions of some arrays.

Kandemir, Ramanujam, and Choudhary (1997) simply mention that one of the main causes for false sharing is the parallelization of a loop carrying

spatial reuse (Li 1993; Wolfe 1996). Kandemir, Ramanujam, and Choudhary (1999) address false sharing by maintaining a list of possible optimizations and discarding those where the outermost parallel loop carries spatial reuse. Of the remaining ones, they pick one where the parallel loop is in the outermost position.

2.4 Lim, Liao, et al. (2001)

In terms of locality optimization, Lim, Liao, et al. (2001) apply affine partitioning (Lim and Lam 1998) to recursively find outermost parallel loops and then identify parallel loops and innermost permutable loops that carry reuse (Wolf and Lam 1991a, see Section 2.1). These loops are strip-mined and the strip-mined parts are moved innermost.

2.5 Kandemir, Ramanujam, Choudhary, and Banerjee (2001)

Kandemir, Ramanujam, Choudhary, and Banerjee (2001) assume a fixed layout and compute a loop nest transformation. In particular, they pick the last column of the inverse transformation matrix Q to be a reuse direction, i.e., from $\ker F$ (1) for self-temporal reuse or from $\ker G$ (3) for self-spatial reuse, ensuring that the innermost transformed loop has temporal or spatial locality. The list of references is ordered according to importance, which is based on profiling information. As many references as possible are optimized for temporal locality and the others for spatial locality. Spatial locality in the second innermost loop can be exploited by taking the second to last column of Q from $\ker G$.

They prefer a last column that is a unit vector. If this is not possible, then they write the general solution as a linear combination of a basis for the solution space and try to find the right coefficients that ensure that the transformed dependence vectors are lexicographically positive. They do this based on a closed-form expression for T that assumes that the last component of the last column of Q is non-zero. It this entry is zero, then they resort to completion techniques of Bik (1996) or Li (1993).

2.6 Bastoul and Feautrier (2003b)

Bastoul and Feautrier (2003b) compute a (partial) schedule for "chunks" (groups of statement instances) that have a limited footprint, i.e., where

$$\operatorname{rank} \begin{bmatrix} T \\ F \end{bmatrix} - \operatorname{rank} T \tag{6}$$

is sufficiently small. This includes the case where self-temporal reuse is captured by the chunks, i.e., where the rows of T are linear combinations of the rows of F and the footprint measure (6) is zero. Of the transformation matrices T with sufficiently small footprint, those with the smallest traffic, measured by

$$\operatorname{rank} T, \tag{7}$$

are considered first. Note, in particular, that such T need not be of full column rank. That is, it typically only specifies a partial schedule. If a matrix T satisfying the rank constraints can be found, then the chunking function is expressed as

$$CT\mathbf{x} + \mathbf{k}, \tag{8}$$

with C a matrix of full row rank. The values of C and $\mathbf{k}$ (one pair for each statement) are computed such that the chunking function satisfies the validity

constraints by applying the Farkas lemma (Feautrier 1992a). Presumably, rows of C are computed one by one, but it is not explained how these rows can be chosen to be linearly independent of each other. Bastoul and Feautrier (2004) (see Section 2.7) provide some more details on this construction.

Self-spatial reuse is taking into account by adding the extra constraint that T should be orthogonal to H inside the algorithm for constructing T. Note, though, that in reality it is sufficient for H to be linearly independent of T and that F should be linearly dependent on T for spatial reuse to be exploited. See constraint (10) below.

Bastoul and Feautrier (2003a) describe the same algorithm, but with fewer details on spatial locality and code generation.

2.7 Bastoul and Feautrier (2004)

Bastoul and Feautrier (2004) offer some more details on the technique of Bastoul and Feautrier (2003a) for obtaining valid (partial) schedules with a prescribed null-space.

For exploiting self-temporal reuse on a single reference, they propose selecting a reuse direction from $\ker F$ and then computing the orthogonal complement of this direction to form the pre-schedule T. The actual schedule is then selected to consist of linear combinations of the rows of T (8). For multiple references, the authors propose to select a reuse direction for each reference and to compute the orthogonal complement of the space spanned by all these directions. Similar expressions can be found for self-spatial reuse, group-temporal reuse and group-spatial reuse (Wolf and Lam 1991a, see Section 2.1).

Each row $\mathbf{c}_i$ and k_i of C and $\mathbf{k}$ is computed in turn by plugging in $\mathbf{c}_i T + k_i$ in the constraints obtained from Farkas, updating T in each step. Solutions where $\mathbf{c}_i$ is a linear combination of the first $i-1$ rows of (the updated) T are removed, but it is not explained how these solutions are removed. A solution with minimal values for $\mathbf{c}_i T$ is selected and the dependences carried by this row are removed. Finally, row i of T is replaced by $\mathbf{c}_i T$. If this linear combination does not involve the original row i of T, then this row is first swapped with a later row that is involved in the linear combination.

Directly incorporating such constraints into the `isl` scheduler (Verdoolaege and Janssens 2017) is not obvious because the scheduling problem is formulated in terms of the original schedule coefficients. However, computing a linear combination of the rows of T is the same as computing a row with a null-space that includes that of T. It is therefore sufficient to add some equality constraints (corresponding to the orthogonal complement of T) on the schedule coefficients.

2.8 Bondhugula, Hartono, et al. (2008)

Bondhugula, Hartono, et al. (2008) present the design and implementation of the `PLuTo` tool, based on the scheduler of Bondhugula, Baskaran, et al. (2008). In terms of locality, their main focus is temporal locality. Bondhugula, Hartono, et al. (2008, Section 5.4) do mention the possibility of optimizing spatial locality by performing interchanges in the intra-tile bands, but do not provide any details. Support for these intra-tile interchanges for spatial locality was made available in version `0.8.1-53-g63b86f2` (2012). According to Zinenko et al. (2017), each schedule dimension is assigned a value

$$n \cdot (2s + 4t + 8v - 16(a - s - t)), \tag{9}$$

with n the number of statements for which this schedule dimension corresponds to a loop, a the total number of accesses, s the number of accesses that exhibit

spatial locality, t the number of accesses that exhibit temporal locality, and v equal to one if the loop is vectorizable. An access is considered to exhibit spatial locality, if the column in $F\,T^{-1}$ corresponding to the schedule dimension has a coefficient between 1 and 4 in the final row and zeros in all other rows. It is considered to exhibit temporal locality if the entire column is zero. The schedule dimension is considered to be vectorizable if it is parallel and if all accesses exhibit either spatial or temporal locality. The schedule dimension with the largest value (9) is sunk.

2.9 Trifunovic et al. (2009)

Trifunovic et al. (2009) exhaustively consider all permutations of loops and then for each statement and for each loop level consider strip-mining at that level (which represents vectorization). For each configuration, a cost is computed and the configuration with the best cost is kept. The cost model takes into account strided accesses and alignment by means of the transformed access functions. There is no mention of any check for the validity of the permutation.

2.10 Vasilache et al. (2012)

Vasilache et al. (2012) define *contiguity* as the innermost transformed loop iterator appearing in at most one dimension of the array index expression.[1] If it is the innermost dimension of the array index expression that exhibits "contiguity", then this is a necessary condition for the innermost transformed loop having spatial locality. In this innermost case, the characterization is

$$\ker T_1 \subseteq \ker G, \tag{10}$$

with

$$T = \begin{bmatrix} T_1 \\ T_2 \end{bmatrix} \tag{11}$$

the transformation matrix and T_2 its final row. This is essentially the same criterion as used by Kandemir, Ramanujam, and Choudhary (1999) (see Section 2.3), except that the latter additionally impose the (for them) unneeded constraint that the innermost transformed loop iterator should appear with coefficient one in the innermost index expression and that, moreover, the innermost index expression should not involve any other transformed loop iterators. These extra constraints actually ensure spatial locality (rather than temporal locality), while the criterion (10) on its own does not. Recall that in case the innermost index expression exhibits temporal locality, the extra constraints are imposed on the next innermost index expression.

Example 1. *Consider an access expression*

$$\{\, \mathtt{S}[i,j,k] \to \mathtt{A}[i,j] \,\} \tag{12}$$

and a schedule

$$\{\, \mathtt{S}[i,j,k] \to [j,i,k] \,\}. \tag{13}$$

That is,

$$G = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \tag{14}$$

[1] This may not be entirely clear from a first reading because there is a typo in the definition in their Section 3.1.1. In particular, it is a *column* of $F\,T^{-1}$ that is required to have a single non-zero entry (as indicated by the transpose) and not a row of this matrix (as suggested by the incorrect subscript expressions $m_{r,i}$).

and

$$T = \left[\begin{array}{c} T_1 \\ \hline T_2 \end{array}\right] = \left[\begin{array}{ccc} 0 & 1 & 0 \\ \hline 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right]. \tag{15}$$

G is clearly a linear combination of the rows of T_1, meaning that $\ker T_1 \subseteq \ker G$. *However, the transformed index expression is*

$$\{\, [a, b, c] \to \mathtt{A}[b, a] \,\}. \tag{16}$$

While there is temporal locality with respect to the innermost transformed loop iterator, the next loop iterator clearly does not exhibit spatial locality.

Even though the scheduler produces a multi-dimensional schedule based on a convex space of valid schedules (Vasilache 2007; Pouchet et al. 2011), it is called several times, each time fixing an additional row of the schedule. If the condition (10) is already satisfied for T_0, the schedule fixed so far, then contiguity is already achieved and no further constraints need to be added. Otherwise, the quantity

$$n = \operatorname{rank} \begin{bmatrix} G \\ T_0 \end{bmatrix} - \operatorname{rank} T_0 \tag{17}$$

is examined and compared to the "number of remaining dimensions to schedule", here denoted as g. This number does not include the final schedule row and is therefore equal to $d - 1 - \operatorname{rank} T_0$ (Vasilache 2017), since condition (10) needs to be satisfied by the part of the schedule that excludes this final row. If $n > g$, then contiguity cannot be achieved. If $n = g$, then the next row is taken to be a linear combination of the rows of G by introducing unconstrained variables $\boldsymbol{\lambda}$ and enforcing

$$\mathbf{c}^{\mathrm{t}} = G\boldsymbol{\lambda}. \tag{18}$$

This equality constraint is encoded as a pair of inequality constraints that are only enforced if the corresponding decision variable is set. If $n < g$, then the next row is apparently chosen such that it is not orthogonal to any of the elements in a basis for

$$\begin{bmatrix} G \\ T_0 \end{bmatrix}. \tag{19}$$

The reason for this choice is not entirely clear.

2.11 Kong et al. (2013)

Kong et al. (2013) define stride-0 and stride-1 accesses as those where the innermost transformed loop iterator does not appear in the innermost array index expression or appears with a coefficient one, without stating explicitly that it should not appear in the outer array index expressions. The sufficient condition for stride-1 accesses that they end up using, does take this into account. In particular, they require

1. that every statement index that appears in an outer array index expression appears in an outer schedule row and
2. that at least one of the statement indices appearing in the innermost array index expression (with coefficient 1) does not appear in outer schedule rows.

On the other hand, this condition does not explicitly state that this statement index should appear in the innermost schedule row. The first part of the condition appears to be used as a criterion for stride-0 accesses, but this is clearly not sufficient.

Example 2. *Consider an access expression*

$$\{\, \mathtt{S}[i,j] \to \mathtt{A}[i,j] \,\} \tag{20}$$

and a schedule

$$\{\, \mathtt{S}[i,j] \to [i+j, i] \,\}. \tag{21}$$

This schedule satisfies the first part of the condition since i appears in the outer schedule row. However, the transformed index expression is

$$\{\, [a,b] \to \mathtt{A}[b, a-b] \,\}, \tag{22}$$

which is clearly neither stride-0 nor stride-1.

Since the authors are aiming for SIMD vectorization, they combine stride-1 (or stride-0) accesses with innermost parallelism. Like Vasilache et al. (2012) (see Section 2.10), they compute a multi-dimensional schedule in one shot (Vasilache 2007; Pouchet et al. 2011) (without calling it multiple times), also taking into account dependence distances and inner permutability. Some details are missing from the description. In particular, there is no mention of linear independence of the schedule rows. According to Kong (2017), linear independence is not enforced, but the final row is required to have at least one non-zero coefficient. The sum of all schedule coefficients is also an optimization criterion, but its position is not mentioned in the list of objectives of Kong et al. (2013, Section 4.5).

The authors introduce two binary decision variables σ_1^F and σ_2^F, where $\sigma_1^F = 1$ appears to ensure that the statement indices that appear in the outer array index expressions also appear in outer schedule rows, while $\sigma_2^F = 1$ apparently tries to ensure that that the statement indices that appear in the innermost array index expression do not appear in outer schedule rows. However, these constraints are imposed by means of binary decision variables $\gamma_{i,j}$ each of which is zero when the corresponding schedule coefficient $\theta_{i,j}$ is zero and is allowed to be one when $\theta_{i,j}$ is not zero. The authors claim that they ensure that $\theta_{i,j}$ is effectively equal to one when $\theta_{i,j}$ is not zero, but they try to achieve this by maximizing $\sum \gamma_{i,j}$. This maximizing $\sum \gamma_{i,j}$ is used as the last optimization criterion and therefore does not force the schedule coefficient $\theta_{i,j}$ to be zero when $\gamma_{i,j}$ has been set to zero by earlier objective functions. There is therefore no guarantee that the selected coefficients do not appear in outer schedule rows when $\sigma_2^F = 1$. In summary, the constraints enforced when $\sigma_1^F = 1$ and $\sigma_2^F = 1$ only appear to create favorable conditions where stride-1 or stride-0 accesses may appear rather than necessarily enforcing stride-1 and/or stride-0 accesses.

2.12 Verdoolaege (2016)

Verdoolaege (2016) describes the operations on binary relations and functions that are used in this report, including

- the inverse A^{-1} of a binary relations A
- the union $A \cup B$ of two binary relations A and B
- the composition $B \circ A$ of two binary relations A and B
- the range product of two functions

2.13 Verdoolaege and Janssens (2017)

Verdoolaege and Janssens (2017) provide a detailed description of the scheduler implemented in `isl` that was first introduced by Verdoolaege, Juega, et al. (2013) and on top of which the support for consecutivity described in this report is added. The scheduler takes as input a set of statement instances that need to be scheduled as well as different forms of schedule constraints. The most important schedule constraints are

- validity schedule constraints, which enforce a relative order between pairs of statement instances,
- proximity schedule constraints, which tell the scheduler to try and schedule pairs of statement instances close to each other, and
- coincidence schedule constraints, which tell the scheduler to try and schedule pairs of statement instances together for as long as possible.

The output of the scheduler is a schedule tree (Verdoolaege, Guelton, et al. 2014) describing the relative order of the statement instances. The main types of nodes in this tree are band nodes and sequence nodes. A band node orders statement instances according to a multi-dimensional affine functions. The individual (single-dimensional) affine functions that form such a multi-dimensional affine functions are called the members of the band. The members of a band are usually permutable. If so, the band is also tilable. A sequence node partitions the statement instances over its children, with the order of the children determining their relative execution order.

Further details on the `isl` scheduler are described in Section 3.3.3, Section 3.3.4 and Section 3.4.1.

2.14 Zinenko et al. (2017)

Zinenko et al. (2017) extend the `isl` scheduler to take into account spatial locality by introducing *spatial* proximity schedule constraints that contain pairs of statement instances that access adjacent array elements. Array elements are considered adjacent if they only differ in the final array dimension and do so by only a small amount, say 4. Spatial proximity schedule constraints are only constructed between references with identical index expressions up to a constant shift in the final index expression. Furthermore, if the index expression is linearly independent of some outer loops, then spatial proximity schedule constraints are only considered between statement instances executed in the same iterations of those outer loops. The spatial proximity schedule constraints are further grouped according to groups of uniformly generated references and sorted according to their rank (number of independent outer loops) and multiplicity. During the computation of a schedule row, the maximal schedule distance between pairs of elements in each group is minimized in turn. Each group with a non-zero distance is removed from consideration for any subsequent schedule rows.

Due to the way spatial proximity schedule constraints are constructed, many of them will have a strong correspondence with the intra-statement consecutivity schedule constraints of this report in the sense that a satisfied intra-statement consecutivity schedule constraint means that the corresponding spatial proximity schedule constraints will have a zero distance in the outer dimensions and a distance of one (in absolute value) in some inner dimension. However, since spatial proximity schedule constraints are not tailored to optimizing consecutivity, they do not distinguish between accessing elements in increasing order and

accessing them in decreasing order. They also make no distinction between directions that should be in outer dimensions and directions that are independent, meaning they will favor putting the independent directions in non-innermost positions, while the handling of intra-statement consecutivity schedule constraints does not result in such a preference. If a choice needs to be made, then this also means that they cannot tell the difference between allowing one of the independent directions to be innermost (while still achieving spatial locality) and allowing one of the outer index expression directions to be innermost (thereby failing to achieve spatial locality). Zinenko et al. (2017) do not consider any preprocessing to combine constraints over multiple arrays, but instead leaves it to the ILP solver to pick the right combination of constraints to solve. It is difficult to predict which approach will produce the best results with the least amount of effort, but one crucial difference is that the handling of spatial proximity schedule constraints requires a sequence of additional variables in the ILP problem for each group of spatial proximity schedule constraints, while the handling of intra-statement consecutivity schedule constraints does not require any additional variables in the ILP problem. Furthermore, the handling of intra-statement consecutivity schedule constraints does not involve minimizing distances between pairs of instances that may in the end turn out to correspond to failed spatial locality constraint.

3 Reference Consecutivity

This section forms the core of this report and consists of four parts, the schedule constraints that are used to communicate consecutivity goals to the scheduler, how these consecutivity schedule constraints can be set by the user exemplified by the `PPCG` implementation, how the consecutivity schedule constraints are translated into constraints on schedule coefficients and how these constraints on schedule coefficients are solved. The main focus is reference consecutivity and not array consecutivity.

3.1 Schedule Constraints

Two types of schedule constraints are introduced for targeting consecutivity: intra-statement consecutivity schedule constraints and inter-statement consecutivity schedule constraints. These two types are first described in abstract terms and then in terms of how they are represented in `isl`.

3.1.1 Intra-statement consecutivity schedule constraint

Consider first the case of a single reference to an array. If the index expression is constant (this is in particular the case when the array is zero-dimensional, i.e., a scalar), then consecutivity is automatic and nothing needs to be done. Otherwise, in order to achieve consecutivity in the innermost index expression of this array reference, the coefficient of the innermost schedule dimension, i.e., the one that prescribes the innermost loop after AST generation, needs to be equal to 1 or 0. Furthermore, this schedule dimension should not appear in any of the outer index expressions. In case the innermost schedule dimension has coefficient 0 in the innermost index expression, the second innermost innermost schedule dimension also needs to appear with a coefficient 1 or 0 in the innermost index expression, while also not appearing in any of the outer index expressions, and so on until a schedule dimension is reached with coefficient 1 in the innermost index expression. Let this schedule dimension correspond to loop i in the generated code. What the conditions above mean is that the loops nested inside this loop

do not affect the accessed element and that an increment of 1 in loop i results in an increment of 1 in the innermost index expression. Since the loop does not affect any other index expressions, this means more specifically that the next element of the array is being accessed.

It may also be useful to consider consecutivity in more than one innermost index expression, especially to handle virtual references that cover multiple references from the same statement, as explained in Section 3.2.1 below. In this case, similar conditions apply to the next group of innermost schedule dimensions and the second innermost index expression. The effect is that an increment of 1 in the corresponding loop in the transformed program results in the next row of the array being accessed. These conditions and the information needed for their evaluation are captured by the following two definitions.

Definition 3 (Intra-statement Consecutivity Schedule Constraint)**.** *An* intra-statement consecutivity schedule constraint *consists of an n-dimensional statement S, a $d \times n$-dimensional reference matrix F, mapping instances of S to elements of a d-dimensional array, and a number f of final rows, with $1 \leq f \leq d$. Alternatively, F may be specified using a $(d-f) \times n$-dimensional matrix G and an $f \times n$-dimensional matrix H, with*

$$F = \begin{bmatrix} G \\ H \end{bmatrix}. \tag{23}$$

Definition 4 (Intra-statement Consecutivity Schedule Constraint Satisfaction)**.** *An intra-statement consecutivity schedule constraint (using the notations of Definition 3) is considered to be* satisfied *by a schedule with linear part T for statement S, if $F\,T^{-1}$ is of the form*

$$\begin{array}{l} d-f \left\{ \right. \\ f \left\{ \right. \end{array} \left[\begin{array}{c|c} & 0 \quad 0 \quad \cdots \quad 0 \\ & \vdots \quad \vdots \quad \quad \vdots \\ A & 0 \quad 0 \quad 0 \quad \cdots \quad 0 \quad 0 \\ & 1 \quad 0 \quad \cdots \\ & \quad\;\; 1 \quad \ddots \quad \vdots \\ & \quad\;\; L \quad \ddots \quad 0 \\ & \quad\quad\quad\;\; 1 \end{array} \right], \tag{24}$$

for some A and L. That is, there is some number $t \geq 0$ such that among the final $t+f$ columns there are t zero columns and the remaining f columns are such that the top $d-f$ rows are zero and the bottom f rows are lower-triangular with 1 on the diagonal.

What the form (24) for the transformed access matrix $F\,T^{-1}$ means is that there are some t schedule dimensions among the final $f+t$ schedule dimensions that do not appear at all in the transformed index expressions and that the remaining f final schedule dimensions appear in order with coefficient 1 in the final index expressions and do not appear in any earlier index expressions. This is a slight generalization of the criteria (4) and (5) of Kandemir, Ramanujam, and Choudhary (1999) (see Section 2.3), where the coefficient 1 is now required for consecutivity and earlier schedule dimensions can also appear in later index expressions (L does not have to be 0 outside of the t columns).

Some intra-statement consecutivity schedule constraints can be removed from consideration from the start as they are impossible to satisfy. The criteria are based on two forms of linear independence.

Definition 5. *An $m \times n$-dimensional matrix M is said to be* linearly independent *if the rows of M are linearly independent, i.e., if* $\operatorname{rank} M = m$.

Definition 6. *An $m_1 \times n$-dimensional matrix M_1 is said to be* linearly independent *of an $m_2 \times n$-dimensional matrix M_2 if there is no linear dependence among the combined rows that is not a linear dependence when restricted to the rows of M_1 (or M_2), i.e., if*

$$\operatorname{rank} \begin{bmatrix} M_1 \\ M_2 \end{bmatrix} = \operatorname{rank} M_1 + \operatorname{rank} M_2. \tag{25}$$

Note that this is a symmetric property, meaning that if M_1 is linearly independent of M_2, then M_2 is also linearly independent of M_1.

Proposition 7. *Using the notation of Definition 3 on the previous page, if H is not linearly independent, then the intra-statement consecutivity constraint cannot be satisfied.*

Proof. Let V be a matrix that selects the f columns in $F\,T^{-1}$ (24) that have a leading one. Then, in case of satisfaction, $H\,T^{-1}\,V$ is lower-triangular with all ones on the diagonal. This means that $\operatorname{rank} H\,T^{-1}\,V = f$, which is impossible if $\operatorname{rank} H < f$. □

Proposition 8. *Using the notation of Definition 3 on the preceding page, if H is not linearly independent of G, then the intra-statement consecutivity constraint cannot be satisfied.*

Proof. Let V be a matrix that selects the f columns in $F\,T^{-1}$ (24) that have a leading one. Then, in case of satisfaction, $G\,T^{-1}\,V$ is all zero, while $H\,T^{-1}\,V$ is lower-triangular with all ones on the diagonal. Any linear combination of the rows of G and H that is not trivial on H is therefore non-zero when multiplied with $T^{-1}\,V$. The linear combination is therefore also non-zero itself, meaning that G is linearly independent of H. □

The matrix G does not need to be linearly independent. It can be freely replaced by any G' such that $G = X\,G'$ for some X without affecting the satisfiability of the constraint, as long as $\operatorname{rank} G' = \operatorname{rank} G$. In particular, G can be replaced by a basis for the rows of G.

Note that a satisfied intra-statement consecutivity schedule constraint does not in itself guarantee array consecutivity for the accessed array. In particular, there may be more than one access to the same array (in the same or a different statement) and even if the array is only accessed through a single reference, then the access may still be strided if the relevant schedule dimension is strided. Such strided schedule dimensions may be the result of a non-unimodular linear schedule T or of strides in the original statement instance set, but they should be fairly rare. Furthermore, if $d > 1$, then array consecutivity may hold for only subsequences of the executed statement instances if $f < d$ or if not all elements in a row are accessed.

Example 9. *Consider the code in Listing 1 on the next page. The code contains a single access to the* `A`*-array with access matrix*

$$F = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}. \tag{26}$$

A linear transformation matrix

$$T = \begin{bmatrix} 1 & 1 \\ -2 & 0 \end{bmatrix}, \qquad T^{-1} = \begin{bmatrix} 0 & 1/2 \\ 1 & 1/2 \end{bmatrix} \tag{27}$$

```
void f(int);

void g(int N,
    __pencil_consecutive int A[restrict static N][N])
{
    for (int i = 0; i < N; ++i)
        for (int j = i; j < N && i + j < N; ++j)
S:          f(A[i + j][j - i]);
}
```

Listing 1: Input file strided.c

```
for (int c0 = 0; c0 < N; c0 += 1)
  for (int c1 = -((c0 + 1) % 2) - c0 + 1; c1 <= 0; c1 += 2)
    f(A[c0][c0 + c1]);
```

Listing 2: Transformed code for the input in Listing 1

results in the transformed access matrix

$$F\,T^{-1} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \tag{28}$$

which satisfies the reference consecutivity constraint and would therefore appear to achieve array consecutivity in at least the final array dimension. However, the second schedule dimension in T has clearly been scaled by a factor of 2, resulting in a strided schedule dimension as also evident from the transformed code shown in Listing 2.

3.1.2 Inter-statement Consecutivity Schedule Constraints

While intra-statement consecutivity schedule constraints can be used to promote consecutive accesses from a given statement, it can also be useful to try and bring accesses from different statements next to each other. In principle, the scheduler could construct an inter-statement consecutivity schedule constraint from each pair of intra-statement consecutivity schedule constraints by looking for pairs of statement instances that access consecutive elements. However, the intra-statement consecutivity schedule constraints do not keep track of which array is involved and may in practice even be derived from accesses to several arrays, as explained in Section 3.2.1 below. For inter-statement consecutivity schedule constraints that do not correspond to the innermost index expressions, the scheduler would need to have access not just to the array identifiers but also to their sizes. Moreover, the user may want to exert further control over exactly which pairs of instances should be involved. The inter-statement consecutivity schedule constraints therefore also keep track of these pairs of statement instances.

Definition 10 (Inter-statement Consecutivity Schedule Constraint)**.** *An* inter-statement consecutivity schedule constraint *consists of*

- *a binary relation R between a pair of distinct statements S_1 and S_2 of dimension n_1 and n_2 respectively, along with*
- *a pair of intra-statement consecutivity schedule constraints on S_1 and S_2 with reference matrices F_1 and F_2 and with equal number of final rows f.*

```
void f(int);

void g(int N,
        __pencil_consecutive int A[restrict static N])
{
        for (int i = 0; i < N; i += 2) {
S:              f(A[i]);
T:              f(A[i + 1]);
        }
}
```

Listing 3: Input file inter.c

Definition 11 (Inter-statement Consecutivity Schedule Constraint Satisfaction)**.** *An inter-statement consecutivity schedule constraint (using the notations from Definition 10 on the preceding page) is considered to be* satisfied *by a schedule T mapping S_1 and S_2 to a common space, with linear part T_1 for statement S_1 and linear part T_2 for statement S_2 if*

- *the intra-statement constraint (S_1, F_1, f) is satisfied by T_1 with the number of zero columns t equal to some t_1,*
- *the intra-statement constraint (S_2, F_2, f) is satisfied by T_2 with the number of zero columns t equal to some t_2,*
- *$t_1 = t_2$, and*
- *for all pairs of instances $(\mathbf{x}_1, \mathbf{x}_2) \in R$ the following holds:*

$$\exists \mathbf{z} : T(\mathbf{x}_2) - T(\mathbf{x}_1) = \begin{bmatrix} \mathbf{0} \\ 1 \\ \mathbf{z} \end{bmatrix}, \tag{29}$$

with $\mathbf{z}$ some arbitrary vector of size $f + t_1 - 1$ that may depend on $\mathbf{x}_1$ and $\mathbf{x}_2$. That is, the position of 1 in the right-hand side corresponds to the schedule dimension that has coefficient 1 in the index expressions that correspond to the first rows of H_1 and H_2.

That is, the purpose of an inter-statement consecutivity schedule constraint is to achieve consecutivity in the loop that achieves intra-statement consecutivity in the first of the f inner index expressions. If consecutivity also needs to be achieved in later index expressions, then this should be handled by separate inter-statement consecutivity schedule constraints. Inner transformed loops that do not appear in the transformed index expressions do not need to be considered to achieve inter-statement consecutivity. This explains why schedule dimensions following the one that determines inter-statement consecutivity may have arbitrary distances.

Example 12. *Consider the code in Listing 3 and assume that inter-statement consecutivity is desired between instances of statements `S` and `T` that access consecutive elements. The binary relation R is of the form*

$$\{\, \mathrm{S}[i] \to \mathrm{T}[i] : 0 \leq i < N \wedge i \bmod 2 = 0 \,\}, \tag{30}$$

while

$$F_1 = F_2 = \begin{bmatrix} 1 \end{bmatrix} \tag{31}$$

and $f = 1$.

3.1.3 Encoding

Intra-statement consecutivity schedule constraints (Definition 3 on page 12) are encoded as multiple affine expression (`isl_multi_aff`) objects and passed to the scheduler as a list of such objects. In particular, each such multiple affine expression is the range product of two expressions corresponding to G and H (in the notation of Definition 3 on page 12) defined over the space of the statement S.

Example 13. *Continuing from Example 9 on page 13, consider once more the code in Listing 1 on page 14. The multiple affine expression for consecutivity in the inner index expression of the single array reference is*

```
{ S[i, j] -> [[(i + j)] -> [(-i + j)]] }
```

Note that the name of the statement is taken into account in this expression, but the name of the array involved is not.

Inter-statement consecutivity schedule constraints (Definition 10 on page 14) are encoded as binary relations (`isl_map` objects) and passed to the scheduler as a list of such relations. In particular, each such relation corresponds to the relation R (in the notation of Definition 10 on page 14), tagged with references to the two intra-statement consecutivity schedule constraints. That is, the tag identifiers are equal to the output identifiers of the referenced `isl_multi_aff` objects.

Example 14. *Continuing from Example 12 on the preceding page, consider once more the code in Listing 3 on the previous page. The binary relation for consecutivity between `S` and `T` is*

```
[N] -> { [S[i] -> I1[]] -> [T[i] -> I2[]] :
         i mod 2 = 0 and 0 <= i <= -2 + N }
```

with corresponding intra-statement consecutivity schedule constraints

```
{ S[i] -> I1[[] -> [i]] }
{ T[i] -> I2[[] -> [i]] }
```

3.2 Setting Consecutivity Constraints

This section described one approach for setting consecutivity constraints. This approach needs the following input,

- a list of arrays for which consecutivity is desired, and
- the array index expressions or access relations from which array index expressions can be extracted. The access relations needs to be "tagged" (Verdoolaege and Janssens 2017, Section 3.2.4) in order to be able to differentiate between multiple accesses to the same array from the same statement.

The construction of inter-statement consecutivity schedule constraints requires the following additional input

- the sizes of the arrays in all but the outermost dimension
- separate read and may-write access relations

```
void transpose(int N,
    __pencil_consecutive float A[restrict static N][N],
    __pencil_consecutive float C[restrict static N][N])
{
    float tmp[N][N];

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
S:          tmp[i][j] = A[i][j];
T:          C[j][i] = tmp[i][j];
        }
}
```

Listing 4: Input file transpose.c

```
float tmp[N][N];
{
  for (int c0 = 0; c0 < N; c0 += 1)
    for (int c1 = 0; c1 < N; c1 += 1)
      tmp[c0][c1] = A[c0][c1];
  for (int c0 = 0; c0 < N; c0 += 1)
    for (int c1 = 0; c1 < N; c1 += 1)
      C[c0][c1] = tmp[c1][c0];
}
```

Listing 5: Transformed code for the input in Listing 4

- a "cut" access relation (Verdoolaege and Janssens 2017, Section 3.1) between statement instances and data elements that cannot pass information across the corresponding instances. This includes must-writes, but can also include explicit or implicit kills (Verdoolaege and Janssens 2017, Section 3.2.3).
- the original schedule

In the `PPCG` implementation, most of this information is derived by `pet`, including the array sizes, the access relations and the original schedule. The set of consecutive arrays are those marked by `__pencil_consecutive` in the source code. The order of the consecutive arrays is determined by `PPCG` based on their dimensions, with higher-dimensional arrays appearing before lower-dimensional arrays. The user can override the set of consecutive arrays and their order by setting the `--consecutive-arrays` command line option.

If there are consecutive arrays, then the statement grouping of Verdoolaege and Janssens (2017, Section 7.4) in `PPCG` is disabled as this grouping prevents grouped statements from being transformed differently.

Example 15. *Consider the code in Listing 4. If enabled, the statement grouping would group the two statements into a single statement prior to calling the scheduler because each instance of the first statement writes to some temporary data structure that is read by the immediately following instance of the second statement. This would mean only one of the two accesses can be made consecutive. With statement grouping disabled, the two statements can be transformed separately and the code shown in Listing 5 can be produced instead. See also Example 29 on page 41.*

3.2.1 Intra-statement Consecutivity Schedule Constraints

The intra-statement consecutivity handling by PPCG only considers consecutivity in the last index expression. However, this does not mean that the number of rows f in Definition 3 on page 12 is always 1 since PPCG tries to combine constraints from multiple references into a single intra-statement consecutivity schedule constraint, which may have a higher number of fixed rows.

Furthermore, PPCG only takes into account array references that satisfy the following constraints:

- Each statement instance accesses a single array element through the reference. That is, the access relation restricted to the reference is single-valued.
- The access expression is purely affine. That is, it does not involve any integer divisions (not quasi-affine) and it is not defined by different expressions over different subsets of the domain (not piecewise). In principle, it would be sufficient to impose this restriction on only the innermost index expressions and to take overapproximations for the outer index expressions.

  Example 16. *An index expression of the form* `A[(i + j) % 4][k]` *is currently ignored, but it could be conservatively handled as an expression of the form* `A[i][j][k]` *for the purpose of intra-statement consecutivity on the innermost index expression.*

- The (linear part of the) innermost index expression is not a linear combination of the (linear parts of the) outer index expressions. If it is a linear combination, then it is impossible to achieve consecutivity because any increment in the innermost index expression necessarily implies a change in an outer index expression.

  Example 17. *An index expression of the form* `A[i][i]` *cannot be made consecutive by only changing the execution order.*

If there are multiple references in a given statement, then PPCG tries to combine as many of those as possible into one or more composite references. Two forms of combination are applied to a pair of intra-statement consecutivity schedule constraints (G_1, H_1) and (G_2, H_2). They both assume that H_i is linearly independent and also linearly independent of G_i. These properties are preserved in the resulting (G, H) pair.

1. If $H_1 = H_2$, then the two constraints can be combined by setting

$$G = \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \qquad H = H_1, \tag{32}$$

   provided $H_1 = H_2$ is linearly independent of G. This choice of G and H implies

$$F_1 = \begin{bmatrix} G_1 \\ H_1 \end{bmatrix} = \begin{bmatrix} X_1 & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} G \\ H \end{bmatrix} = \begin{bmatrix} X_1 & 0 \\ 0 & I \end{bmatrix} F \tag{33}$$

   and

$$F_2 = \begin{bmatrix} G_2 \\ H_2 \end{bmatrix} = \begin{bmatrix} X_2 & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} G \\ H \end{bmatrix} = \begin{bmatrix} X_2 & 0 \\ 0 & I \end{bmatrix} F, \tag{34}$$

   with X_1 and X_2 some matrices and I the identity matrix. This in turn implies that if (24) is satisfied for F, then it is also satisfied for F_1 and F_2 because X_1 and X_2 only modify the top part of A in (24). Note once more that G can safely be replaced by a basis for the rows of G.

```
void f(int N,
    __pencil_consecutive float A[restrict static N][N],
    __pencil_consecutive float B[restrict static N][N],
    float C[restrict static N][N])
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j <= i; ++j)
            C[i][j] = A[i][i - j] + B[j][i - j];
}
```

Listing 6: Input file dependent.c

Example 18. *To see the need for $H_1 = H_2$ being linearly independent of G, consider the code in Listing 6. The constraint for the `A`-array is of the form*

$$F_1 = \left[\begin{array}{c|cc} G_1 & & \\ \hline H_1 & & \end{array}\right] = \left[\begin{array}{cc|c} 1 & 0 & \\ \hline 1 & -1 & \end{array}\right], \tag{35}$$

while that of `B` is of the form

$$F_2 = \left[\begin{array}{c|cc} G_2 & & \\ \hline H_2 & & \end{array}\right] = \left[\begin{array}{cc|c} 0 & 1 & \\ \hline 1 & -1 & \end{array}\right]. \tag{36}$$

Combining these two constraints despite the linear dependence would result in

$$F = \left[\begin{array}{c|cc} G & & \\ \hline H & & \end{array}\right] = \left[\begin{array}{cc|c} 1 & 0 & \\ 0 & 1 & \\ \hline 1 & -1 & \end{array}\right]. \tag{37}$$

As per Proposition 8 on page 13, this intra-statement consecutivity constraint cannot be satisfied. It is, however, possible to satisfy either F_1 or F_2.

2. If H_2 is linearly independent of

$$\begin{bmatrix} F_1 \\ G_2 \end{bmatrix}, \tag{38}$$

then the two constraints can be combined by setting

$$G = \begin{bmatrix} G_1 \\ G_2 \setminus F_1 \end{bmatrix} \qquad H = \begin{bmatrix} H_1 \\ H_2 \end{bmatrix}, \tag{39}$$

with $A \setminus B$ a matrix C that is linearly independent of B and that is such that the rows of

$$\begin{bmatrix} A \\ B \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} C \\ B \end{bmatrix} \tag{40}$$

span the same space. One way of computing such a matrix C is described in Section 5.4. Since H_2 is linearly independent of F_1 and G_2 combined, it is also linearly independent of H_1. Combined with the fact that both H_1 and H_2 are linearly independent, this means that H is linearly independent. To see that H is also linearly independent of G, note first that

$$\operatorname{rank} \begin{bmatrix} G_1 \\ G_2 \setminus F_1 \\ H_1 \\ H_2 \end{bmatrix} = \operatorname{rank} G_1 + \operatorname{rank} H_1 + \operatorname{rank}(G_2 \setminus F_1) + \operatorname{rank} H_2. \tag{41}$$

H_2 can be split off first because it is linearly independent of the remaining rows by assumption. $G_2 \setminus F_1$ can then be split off because it is linearly independent of F_1 by construction. Finally, H_1 can be split off because it is linearly independent of G_1. Similarly,

$$\operatorname{rank} \begin{bmatrix} H_1 \\ H_2 \end{bmatrix} = \operatorname{rank} H_1 + \operatorname{rank} H_2 \tag{42}$$

because H_2 is linearly independent H_1, while

$$\operatorname{rank} \begin{bmatrix} G_1 \\ G_2 \setminus F_1 \end{bmatrix} = \operatorname{rank} G_1 + \operatorname{rank}(G_2 \setminus F_1) \tag{43}$$

because $(G_2 \setminus F_1)$ is linearly independent of G_1. This means

$$\operatorname{rank} \begin{bmatrix} G_1 \\ G_2 \setminus F_1 \\ H_1 \\ H_2 \end{bmatrix} = \operatorname{rank} \begin{bmatrix} G_1 \\ G_2 \setminus F_1 \end{bmatrix} + \operatorname{rank} \begin{bmatrix} H_1 \\ H_2 \end{bmatrix}. \tag{44}$$

The choice of G above means that G_1 is a linear combination of G, while G_2 is a linear combination of

$$\begin{bmatrix} G \\ H_1 \end{bmatrix}. \tag{45}$$

The choice of G and H therefore implies

$$F_1 = \begin{bmatrix} G_1 \\ H_1 \end{bmatrix} = \begin{bmatrix} X_1 & 0 & 0 \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} G \\ H_1 \\ H_2 \end{bmatrix} = \begin{bmatrix} X_1 & 0 & 0 \\ 0 & I & 0 \end{bmatrix} F \tag{46}$$

and

$$F_2 = \begin{bmatrix} G_2 \\ H_2 \end{bmatrix} = \begin{bmatrix} X_2 & X_3 & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} G \\ H_1 \\ H_2 \end{bmatrix} = \begin{bmatrix} X_2 & X_3 & 0 \\ 0 & 0 & I \end{bmatrix} F, \tag{47}$$

for some X_1, X_2 and X_3. This in turn implies that if (24) is satisfied for F, then it is also satisfied for F_1 and F_2. In the case of F_1, the bottom $f_2 = \operatorname{rank} H_2$ rows of (24) are removed and a linear transformation is applied to the top part of A, preserving the shape with respect to the now $f_1 = \operatorname{rank} H_1$ final rows. In the case of F_1, X_3 may mix in the first f_1 of the final $f = f_1 + f_2$ rows in the remaining rows, but with respect to the final f_2 rows, the shape is preserved.

The final result is a list of possibly composite references with those that cover more original references appearing before those that cover fewer.

Example 19. *Consider the code in Listing 7 on the next page. The single statement contains three accesses to arrays that should all be accessed consecutively. The constraints for the individual accesses are as follows:*

$$F_{\mathtt{A}} = \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 0 & 1 \end{array} \right] \qquad F_{\mathtt{B}} = \left[\begin{array}{cc|c} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array} \right] \qquad F_{\mathtt{C}} = \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array} \right]. \tag{48}$$

Since $H_{\mathtt{B}} = H_{\mathtt{C}}$ is linearly independent of the combination of $G_{\mathtt{B}}$ and $G_{\mathtt{C}}$, the first form of combination can be applied, resulting in

$$F_{\mathtt{BC}} = \left[\begin{array}{ccc} 0 & 0 & 1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \end{array} \right]. \tag{49}$$

```
void matmul(int N, int M, int K,
    __pencil_consecutive float A[restrict static N][K],
    __pencil_consecutive float B[restrict static K][M],
    __pencil_consecutive float C[restrict static N][M])
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < M; ++j)
            for (int k = 0; k < K; ++k)
                C[i][j] += A[i][k] * B[k][j];
}
```

Listing 7: Input file matmul.c

```
for (int c0 = 0; c0 < N; c0 += 1)
  for (int c1 = 0; c1 < K; c1 += 1)
    for (int c2 = 0; c2 < M; c2 += 1)
      C[c0][c2] += (A[c0][c1] * B[c1][c2]);
```

Listing 8: Transformed code for the input in Listing 7

Now, $H_{\mathtt{BC}}$ *is linearly independent of*

$$\begin{bmatrix} F_{\mathtt{A}} \\ \hline G_{\mathtt{BC}} \end{bmatrix} = \left[\begin{array}{ccc} 1 & 0 & 0 \\ \hline 0 & 0 & 1 \\ 0 & 0 & 1 \\ \hline 1 & 0 & 0 \end{array}\right] \tag{50}$$

and so the second form of combination can be applied. In this case, $G_{\mathtt{BC}}$ *is a linear transformation of* $F_{\mathtt{A}}$ *and so* $G_{\mathtt{BC}} \setminus F_{\mathtt{A}}$ *has zero rows. The result of the combination is therefore*

$$F_{\mathtt{ABC}} = \left[\begin{array}{ccc} 1 & 0 & 0 \\ \hline 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right]. \tag{51}$$

Satisfaction of this single constraint ensures consecutivity for all three accesses and corresponds to the loop order (i, k, j)*. The transformed code satisfying the constraint is shown in Listing 8. Note that for the accesses to* `B` *and* `C`*, the innermost loop iterator only appears in the last index expression and does so with coefficient one. For the access to* `A`*, the innermost loop iterator does not appear at all in the index expressions, while the second innermost loop iterator only appears in the last index expression and does so with coefficient one.*

This mechanism for combining information from different array references is similar to the way Kandemir, Ramanujam, and Choudhary (1999) (see Section 2.3) take multiple array references into account. However, they derive additional elements of the inverse transformation matrix directly by examining each array reference in turn, while the mechanism described in this section first collects information from multiple array references into one or more composite array references that are then later used as a whole during the schedule construction.

If there are multiple references to the *same* array from a given statement, then these multiple references are first combined into one or more references that combine the constraints imposed by each individual reference using the

```
void f(int N,
    __pencil_consecutive float A[restrict static N],
    __pencil_consecutive float B[restrict static N])
{
    for (int i = 0; i < N; ++i)
        A[N - i - 1] = B[i] + B[N - i - 1];
}
```

Listing 9: Input file conflict.c

process described above. If this process fails to produce any result that covers all of the original references to the array, then all references to the array from this statement are ignored. This includes the case where one or more of the references fails to satisfy the conditions at the top of this section as in that case the combined reference would not cover the references that have been ignored. The reason for dropping references to such arrays from consideration is that a failure to construct a combined reference points to conflicts in the references for the array. It will therefore not be possible to achieve array consecutivity on this array.

Example 20. *Consider the code in Listing 9. The consecutivity constraints for the two accesses to* `B` *clearly conflict, one requiring the access order to remain the same and the other requiring the access order to be reversed. The array* `B` *is therefore removed from consideration and the intra-statement consecutivity schedule constraint for this statement is only based on the access to* `A`*, which requires the access order to be reversed.*

3.2.2 Inter-statement Consecutivity Constraints

The main choice in constructing inter-statement consecutivity schedule constraints is deciding which pairs of statement instances should be taken into account for consecutivity. The implementation in `PPCG` makes the following assumptions:

- Array references that access array elements more than once should not be considered for inter-statement consecutivity. If an array element is accessed more than once, then it is not clear if all these accesses should be made to occur next to the access(es) to the previous element or rather that the accesses should somehow be paired off with accesses to the previous elements and that only these individual pairs should be executed next to each other. Array references that are not injective are therefore removed from consideration.

- Read accesses should only be made consecutive to other read accesses, while write accesses should only be made consecutive to other write accesses. That is, read accesses should not be made consecutive to write accesses since accesses of different types could not be combined into bursts anyway.

- A pair of accesses to consecutive elements with an intermediate cut to either of the two elements (in the original program) should not be made to be executed consecutively. If there is such an intermediate cut, then the two separated accesses belong to separate "live-ranges" of the pair of accesses and should therefore be considered to be unrelated.

```
void unroll(int N, int M,
    float A[restrict static N][M],
    __pencil_consecutive float B[restrict static M][N])
{
    float t[N][M];
    __pencil_assume(N % 2 == 0);
    __pencil_assume(M % 2 == 0);
    for (int i = 0; i < N; i += 2) {
        for (int j = 0; j < M; j += 2) {
S00:        B[j + 0][i + 0] = A[i + 0][j + 0];
S01:        B[j + 1][i + 0] = A[i + 0][j + 1];
S10:        B[j + 0][i + 1] = A[i + 1][j + 0];
S11:        B[j + 1][i + 1] = A[i + 1][j + 1];
        }
    }
}
```

Listing 10: Input file unrolled.c

In contrast to intra-statement consecutivity, which is only considered at the innermost index expression in `PPCG`, inter-statement consecutivity is considered at every index expression in `PPCG`, starting from the innermost index expression. In particular, if D is the maximal dimension of the arrays marked consecutive, then inter-statement consecutivity is considered at the i-th innermost index expression for $1 \leq i \leq D$.

An auxiliary relation used in the construction of inter-statement consecutivity schedule constraints is the next-array-element relation at the i-th innermost position, N_i. For each (consecutive) array A of dimension d greater than or equal to i and of size $\mathbf{S}$, the relation N_i contains pairs of elements

$$A[\mathbf{x}] \to A[\mathbf{y}] \tag{52}$$

with

$$x_j = S_j \quad \text{if } j > d - i - i \tag{53}$$

and with

$$y_j = \begin{cases} x_j & \text{if } j < d - i - i \\ x_j + 1 & \text{if } j = d - i - i \\ 0 & \text{if } j > d - i - i \end{cases} \tag{54}$$

Example 21. *Consider the code in Listing 10. Only array `B` has been marked as consecutive. The next-array-element relation for the innermost index expression is therefore*

$$N_1 = \{\, \mathtt{B}[x_1, x_2] \to \mathtt{B}[x_1, x_2 + 1] \,\}, \tag{55}$$

while for the second innermost index expression, it is

$$N_2 = \{\, \mathtt{B}[x_1, N - 1] \to \mathtt{B}[x_1 + 1, 0] \,\}. \tag{56}$$

For each index expression position, starting from innermost, each read is then matched to the reads that access the next element, while each write is matched to the writes that access the next element. However, only those pairs should be kept that have no intermediate cut access (in the original schedule). Let A be either the read access relation or the may-write access relation, let C be the cut access relation and let S be the original schedule. The relation between statement

instances in the domain of A that access consecutive elements is computed using the dependence analysis engine of `isl` (Verdoolaege and Janssens 2017, Section 3.1) with the following input:

- sink access relation: A
- may-source access relation: $N_i \circ A$
- cut access relation: $C \cup (N_i \circ C)$
- schedule: S

Note that since all the access relations are tagged, the schedule S needs to be tagged as well. The resulting may-dependence relation $D_{i,1}$ contains pairs of may-source and sink instances such that the may-source instance accesses an element for which the next element is the one that is accessed by the sink instance, with the restriction that there are no intermediate cuts to either element. The elements may also be accessed in reverse order in the original program and so a second call to the dependence analysis engine is needed with inputs

- sink access relation: $N_i \circ A$
- may-source access relation: A
- cut access relation: $C \cup (N_i \circ C)$
- schedule: S

The resulting may-dependence relation $D_{i,2}$ maps accesses to later accesses to the previous element. The final result is

$$D_i = D_{i,1} \cup D_{i,2}^{-1}. \tag{57}$$

Note that if the cut access relation C is empty (on the consecutive arrays), then the two calls to the dependence analysis engine can be replaced by the simple computation

$$D_i = (N_i \circ A)^{-1} \circ A. \tag{58}$$

For each pair of statement-reference pairs in D_i from distinct statements, an inter-statement consecutivity schedule constraint is then constructed with R the corresponding subset of D_i, the reference matrices F_1 and F_2 derived from the corresponding index expressions and the number of final rows f set to i.

Example 22. *Continuing from Example 21 on the preceding page, the write access relation W (restricted to `B`) is*

$$\begin{aligned}
&\{\, \mathtt{S10}[i,j] \to \mathtt{B}[j,1+i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
&\qquad 0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\} \cup \\
&\{\, \mathtt{S00}[i,j] \to \mathtt{B}[j,i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
&\qquad 0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\} \cup \\
&\{\, \mathtt{S01}[i,j] \to \mathtt{B}[1+j,1+i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
&\qquad 0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\} \cup \\
&\{\, \mathtt{S11}[i,j] \to \mathtt{B}[1+j,i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
&\qquad 0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\}.
\end{aligned} \tag{59}$$

Note that since each statement contains only a single references, the reference tags have been omitted. The access to the next element $N_1 \circ W$ is

$$\begin{aligned}
\{\, \mathtt{S10}[i,j] \to \mathtt{B}[j, 2+i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\} \cup \\
\{\, \mathtt{S00}[i,j] \to \mathtt{B}[j, 1+i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\} \cup \\
\{\, \mathtt{S01}[i,j] \to \mathtt{B}[1+j, 2+i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\} \cup \\
\{\, \mathtt{S11}[i,j] \to \mathtt{B}[1+j, 1+i] : (j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge \\
0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\}.
\end{aligned} \tag{60}$$

Since all writes are must-writes in this example, the cut access relation is equal to $W \cup (N_1 \circ W)$. The dependences from $N_1 \circ W$ (source) to W (sink), i.e., $D_{1,1}$, are

$$\begin{aligned}
\{\, \mathtt{S01}[i,j] \to \mathtt{S11}[i' = i, j' = j] : (i) \bmod 2 = 0 \wedge (j) \bmod 2 = 0 \wedge \\
0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\} \cup \\
\{\, \mathtt{S11}[i,j] \to \mathtt{S01}[i' = 2+i, j' = j] : (i) \bmod 2 = 0 \wedge (j) \bmod 2 = 0 \wedge \\
0 \le i \le -4+N \wedge 0 \le j \le -2+M \,\} \cup \\
\{\, \mathtt{S10}[i,j] \to \mathtt{S00}[i' = 2+i, j' = j] : (i) \bmod 2 = 0 \wedge (j) \bmod 2 = 0 \wedge \\
0 \le i \le -4+N \wedge 0 \le j \le -2+M \,\} \cup \\
\{\, \mathtt{S00}[i,j] \to \mathtt{S10}[i' = i, j' = j] : (i) \bmod 2 = 0 \wedge (j) \bmod 2 = 0 \wedge \\
0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\}.
\end{aligned} \tag{61}$$

There are no dependences from W (source) to $N_1 \circ W$ (sink). That is, $D_{1,2}$ is empty. For each of the disjuncts in (61)*, an inter-statement consecutivity schedule constraint is created with fixed rows $f = 1$. For example, the first disjunct yields*

$$\begin{aligned}
\{\, [\mathrm{S01}[i,j] \to \mathrm{S011}[]] \to [\mathrm{S11}[i' = i, j' = j] \to \mathrm{S111}[]] : \\
(j) \bmod 2 = 0 \wedge (i) \bmod 2 = 0 \wedge 0 \le i \le -2+N \wedge 0 \le j \le -2+M \,\},
\end{aligned} \tag{62}$$

with corresponding intra-statement consecutivity schedule constraints

$$\begin{aligned}
\{\, \mathrm{S01}[i,j] \to \mathrm{S011}[[j] \to [i]] \,\} \\
\{\, \mathrm{S11}[i,j] \to \mathrm{S111}[[j] \to [i]] \,\}
\end{aligned} \tag{63}$$

At the second innermost index expression, the access to the next element $N_2 \circ W$ is

$$\begin{aligned}
\{\, \mathtt{S10}[i = -2+N, j] \to \mathtt{B}[1+j, 0] : (j) \bmod 2 = 0 \wedge (N) \bmod 2 = 0 \wedge \\
N \ge 2 \wedge M > 0 \wedge 0 \le j \le -2+M \,\} \cup \\
\{\, \mathtt{S11}[i = -2+N, j] \to \mathtt{B}[2+j, 0] : (j) \bmod 2 = 0 \wedge (N) \bmod 2 = 0 \wedge \\
N \ge 2 \wedge M > 0 \wedge 0 \le j \le -2+M \,\}
\end{aligned} \tag{64}$$

Again, the cut access relation is equal to $W \cup (N_2 \circ W)$ in this example. The dependences from $N_2 \circ W$ (source) to W (sink), i.e., $D_{2,1}$, are

$$\begin{aligned}
\{\, \mathtt{S11}[i = 0, j] \to \mathtt{S00}[i' = 0, j' = 2+j] : N = 2 \wedge (j) \bmod 2 = 0 \wedge \\
0 \le j \le -4+M \,\}.
\end{aligned} \tag{65}$$

The reverse dependences from W (source) to $N_2 \circ W$ (sink), i.e., $D_{2,2}^{-1}$, are

$$\begin{array}{l}\{\,\mathtt{S11}[i=-2+N,j] \to \mathtt{S00}[i'=0,j'=2+j]: \\ \quad (N) \bmod 2 = 0 \wedge (j) \bmod 2 = 0 \wedge N \geq 4 \wedge 0 \leq j \leq -4+M\,\} \cup \\ \{\,\mathtt{S10}[i=-2+N,j] \to \mathtt{S01}[i'=0,j'=j]: \\ \quad (N) \bmod 2 = 0 \wedge (j) \bmod 2 = 0 \wedge N \geq 3 \wedge 0 \leq j \leq -2+M\,\} \cup \\ \{\,\mathtt{S10}[i=0,j] \to \mathtt{S01}[i'=0,j'=j]: \\ \quad N = 2 \wedge (j) \bmod 2 = 0 \wedge 0 \leq j \leq -2+M\,\}.\end{array} \tag{66}$$

For each pair of statements (in general, for each pair of accesses from distinct statements), an inter-statement consecutivity schedule constraint is created with fixed rows $f = 2$. For example, for the pair $(\mathtt{S10}, \mathtt{S01})$, it is of the form

$$\begin{array}{l}\{\,[\mathtt{S10}[i=-2+N,j] \to \mathrm{S102}[]] \to [\mathtt{S01}[i'=0,j'=j] \to \mathrm{S012}[]]: \\ \quad (j) \bmod 2 = 0 \wedge (N) \bmod 2 = 0 \wedge N \geq 2 \wedge 0 \leq j \leq -2+M\,\},\end{array} \tag{67}$$

with corresponding intra-statement consecutivity schedule constraints

$$\begin{array}{l}\{\,\mathrm{S10}[i,j] \to \mathrm{S102}[[] \to [j,i]]\,\} \\ \{\,\mathrm{S01}[i,j] \to \mathrm{S012}[[] \to [j,i]]\,\}\end{array} \tag{68}$$

3.3 Handling Consecutivity Constraints

This section describes how consecutivity constraints are transformed to constraints on the schedule coefficients during scheduling. The way these constraints on the schedule coefficients are solved is described in Section 3.4 below.

3.3.1 Intra-statement Consecutivity Schedule Constraints

The following proposition shows the properties the linear part of a schedule needs to have for it to satisfy an intra-statement consecutivity schedule constraint. For simplicity, it does so for the case where all the t zero columns of Definition 4 on page 12 appear at the end. If they appear in other positions, then the same conditions apply because the rows of T and the corresponding columns of $Q = T^{-1}$ can simply be permuted in the right position.

Proposition 23. *Given an intra-statement consecutivity schedule constraint with the notations of Definition 3 on page 12. Let T be the $n \times n$ linear part of the schedule for statement S, i.e., in particular with T linearly independent. Then T satisfies the consecutivity constraint (Definition 4 on page 12) with the t zero columns in $F\,T^{-1}$ (24) at the end iff T can be subdivided as*

$$T = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} \tag{69}$$

with T_3 a $t \times n$-dimensional matrix, T_2 a $f \times n$-dimensional matrix and T_1 a $(n-f-t) \times n$-dimensional matrix such that

- $\ker T_1 \subseteq \ker G$
- *each row $T_{2,k}$ of T_2 can be written as*

$$T_{2,k} = H_k + \mathbf{b}^{\mathrm{t}} T_1 + \sum_{j:1 \leq j < k} d_j T_{2,j} \tag{70}$$

for some $\mathbf{b}$ and $\mathbf{d}$.

Proof. Subdivide the columns of $Q = T^{-1}$ accordingly, i.e.,

$$Q = \begin{bmatrix} Q_1 & Q_2 & Q_3 \end{bmatrix}. \tag{71}$$

Note that the inverse exists because T is linearly independent. The columns of Q_2 and Q_3 form a basis for the kernel of T_1. As per (24), these columns also need to be in the kernel of G, showing that $\ker T_1 \subseteq \ker G$.

Since T is linearly independent, H_k can be written as a linear combination of the rows of T,

$$H_k = \mathbf{b}_1^{\mathrm{t}} T_1 + \mathbf{b}_2^{\mathrm{t}} T_2 + \mathbf{b}_3^{\mathrm{t}} T_3. \tag{72}$$

Multiplying this equation on the right with a column from Q_3 results in 0 on the left hand side as per (24) and the corresponding element from $\mathbf{b}_3$ on the right hand side. Therefore, $\mathbf{b}_3$ is zero. The same holds for the elements of $\mathbf{b}_2$ after k, while for element k, the left hand side is 1 and so this element of $\mathbf{b}_2$ needs to be equal to 1. This shows that $T_{2,k}$ needs to be of the form (70).

Conversely, a linearly independent matrix T satisfying the conditions of the proposition results in an $F\,T^{-1}$ of the form (24) and therefore satisfies the consecutivity constraint. □

This criterion is an extension of criterion (10) of Vasilache et al. (2012) (see Section 2.10) that is specialized from spatial locality to consecutivity. Note that $\ker T_1 \subseteq \ker G$ is the same as saying that G needs to be a linear combination of T_1. The strategy is therefore to first construct schedule rows with linear parts that are linear combinations of the rows of G. As soon as $\operatorname{rank} G$ linearly independent such rows have been found, G will also be a linear combination of T_1. This strategy is similar to those of Bastoul and Feautrier (2004) (see Section 2.7) and Vasilache et al. (2012) (see Section 2.10), but the mechanism for obtaining such linear combinations (described below in Section 3.4.2) is different. Once a suitable T_1 has been found, the linear parts of the next rows (T_2) are set equal to successive rows of H (plus a linear combination of earlier schedule rows).

Let T_0 be the linear part of the schedule computed so far. In order to ensure linear independence of T, the scheduler makes sure that the next row is linearly independent of T_0 (Verdoolaege and Janssens 2017, Section 6.5.6). In order to ensure that a suitable choice satisfying (70) can still be made in subsequent steps, this next row also needs to be linearly independent of (the remaining rows of) H. In fact, the next row needs to be linearly independent of T_0 and H combined. Imposing this linear independence is especially important during the phase where T_1 is constructed to have the rows of G as linear combinations. The rows of T_2 are made equal to successive rows of H and are therefore linearly independent of T_0 by construction since T_1 was constructed to be linearly independent of H and H itself is linearly independent.

If $\operatorname{rank} F < n$, then T necessarily contains additional rows that are linearly independent of F. These rows may be inserted at any position. However, if they are inserted after the row that is equal to the first row of H (plus some linear combination), then they should not be used in the linear combinations added to schedule rows made equal to successive rows of H. In the notation of the proof of Proposition 23, this ensures that $\mathbf{b}_3$ remains zero.

Example 24. *To illustrate the insertion of linearly independent rows, consider an index expression of the form*

$$\{\, \mathtt{S}[i,j,k] \to \mathtt{A}[i,j] \,\} \tag{73}$$

and assume that both index expressions need to be made consecutive, i.e., the intra-statement consecutivity schedule constraint is of the form

$$\{\, \mathtt{S}[i,j,k] \to [[] \to [i,j]] \,\}. \tag{74}$$

Let a, b and c be the outer loop iterators of the generated code, then different positions for simply inserting a k-row in the schedule result in the following accesses

$$\begin{array}{lcl} i,j,k & \to & \mathtt{A}[a][b] \\ i,k,j & \to & \mathtt{A}[a][c] \\ k,i,j & \to & \mathtt{A}[b][c] \end{array} \tag{75}$$

In all these cases, the generated index expression is consecutive in the innermost loop iterators that affect the index expression. When the linearly independent row is used in the linear combinations of subsequent rows, then the following cases arise:

$$\begin{array}{lcl} i,k,j+k & \to & \mathtt{A}[a][c-b] \\ k,i,j+k & \to & \mathtt{A}[b][c-a] \end{array} \tag{76}$$

In the last case, the linearly independent row is inserted before the first row that corresponds to the first row of H. The generated index expression is still consecutive in the innermost loop iterators that affect the index expression. In the first case, the linearly independent row is inserted after *the first row that corresponds to the first row of H. In this case, the generated index expression is no longer consecutive in the innermost loop iterators that affect the index expression since the intermediate b-loop affects the last index expression in a non-consecutive way.*

If the statement dimension n is smaller than the maximal statement dimension m, then the default scheduler drops the constraint of linear independence from T_0 as long as a total number of n linearly independent rows can still be found by subsequent steps. In the presence of intra-statement consecutivity schedule constraints, the case of linear dependence on previous rows needs to be allowed as an explicit, separate case because in the converse case (where the new row is linearly independent), the regular constraints on the schedule coefficients need to be imposed.

Table 11 on the next page summarizes the constraints that are introduced on the linear part $\mathbf{c}$ of the next schedule row. As long as

$$r_1 = \operatorname{rank} T_0 < \operatorname{rank} \begin{bmatrix} T_0 \\ G \end{bmatrix} = r_2, \tag{77}$$

G cannot be written as a linear combination of T_0 yet and the next row is taken to be a linear combination of G and T_0, without also being a linear combination of H and T_0. As soon as $r_1 = r_2$, the next phase starts and the next schedule rows are taken to be equal to successive rows of H, allowing for previous rows of H as well as rows from the first phase to be mixed in, but ensuring that the result is not a linear combination of T_0 and the next rows of H. In both phases, the schedule row is also allowed to be either linearly independent of T_0 and F or to be a linear combination of the previous rows. As soon as all rows of H have been handled, i.e., $h = f$, intra-statement consecutivity has been achieved and no more constraints need to be introduced on the next schedule rows. If at any stage

$$\operatorname{rank} \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} < r_2 + (f - h) \tag{78}$$

then intra-statement consecutivity can no longer be achieved and no more constraints are added either.

Note that as soon as

$$\operatorname{rank} \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} = n, \tag{79}$$

Constraints introduced when $r_1 < r_2$
$\left(\mathbf{c}^{\mathrm{t}} = \mathbf{x}^{\mathrm{t}} \begin{bmatrix} T_0 \\ G \end{bmatrix} \wedge \mathbf{c}^{\mathrm{t}} \neq \mathbf{y}^{\mathrm{t}} \begin{bmatrix} T_0 \\ H \end{bmatrix}\right) \vee \mathbf{c}^{\mathrm{t}} \neq \mathbf{y}^{\mathrm{t}} \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} \vee \mathbf{c}^{\mathrm{t}} = \mathbf{x}^{\mathrm{t}} T_0$
Constraints introduced when $r_1 = r_2 \wedge h < f$
$\mathbf{c}^{\mathrm{t}} = H_h + \mathbf{x}^{\mathrm{t}} \begin{bmatrix} T_1 \\ H_{<h} \end{bmatrix} \qquad \vee \mathbf{c}^{\mathrm{t}} \neq \mathbf{y}^{\mathrm{t}} \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} \vee \mathbf{c}^{\mathrm{t}} = \mathbf{x}^{\mathrm{t}} T_0$
Constraints introduced when $h = f$
none

Table 11: Constraints on (the linear part of) the next schedule row $\mathbf{c}^{\mathrm{t}}$ introduced at different stages of the consecutivity constraints handling process. A constraint of the form $\mathbf{c}^{\mathrm{t}} = \mathbf{x}^{\mathrm{t}} A$ is short for saying that $\mathbf{c}$ should be a linear combination of the rows of A. A constraint of the form $\mathbf{c}^{\mathrm{t}} \neq \mathbf{y}^{\mathrm{t}} A$ is short for saying that $\mathbf{c}$ should *not* be a linear combination of the rows of A, i.e., should be linearly independent of the rows of A. T_0 is the linear part of the schedule computed so far. $r_1 = \operatorname{rank} T_0$. $r_2 = \operatorname{rank} \begin{bmatrix} T_0 \\ G \end{bmatrix}$. h is the number of rows in T_0 made equal to rows of H. T_1 is the part of T_0 before the row that is equal to the first row of H. H_h is row h of H. $H_{<h}$ are the rows of H before row h.

$\mathbf{c}$ is necessarily a linear combination of the rows in that matrix and the case

$$\mathbf{c}^{\mathrm{t}} \neq \mathbf{y}^{\mathrm{t}} \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} \tag{80}$$

in Table 11 no longer needs to be considered. Similarly, if the next row is required to be linearly independent of the previous rows, then the case

$$\mathbf{c}^{\mathrm{t}} = \mathbf{x}^{\mathrm{t}} T_0 \tag{81}$$

does not need to be considered. Recall that the next row is only allowed to be a linear combination of previous rows if the statement is not of maximal dimension and if there is some slack left, in particular when

$$n - r_1 < m - \ell, \tag{82}$$

with n the dimension of the statement, m the maximal statement dimension and ℓ the number of rows in T_0.

If multiple intra-statement consecutivity schedule constraints were specified for the same statement, then a constraint on the schedule coefficients is constructed for each intra-statement consecutivity schedule constraint according to the rules in Table 11. However, the solver is instructed to first try and satisfy the constraint on the schedule coefficients corresponding to the first intra-statement consecutivity schedule constraint on the statement and to only consider the one corresponding to a later intra-statement consecutivity schedule constraint when the one corresponding to the previous intra-statement consecutivity schedule constraint cannot be satisfied. Disjuncts that also appear in the constraint on the schedule coefficients corresponding to previous intra-statement consecutivity schedule constraints are therefore dropped since they are already known to

be unsatisfiable by the time they would be reconsidered. In particular, the linear dependence disjunct (81) is independent of the intra-statement consecutivity schedule constraint and is therefore only considered for the first intra-statement consecutivity schedule constraint. If all disjuncts corresponding to an intra-statement consecutivity schedule constraint are duplicates of disjuncts corresponding to previous intra-statement consecutivity schedule constraints on the same statement, then the entire disjunction is dropped.

3.3.2 Inter-statement Consecutivity Schedule Constraints

Inter-statement consecutivity schedule constraints are handled by treating the two referenced intra-statement consecutivity schedule constraints as regular intra-statement consecutivity schedule constraints and then adding additional constraints depending on the state in the handling of those intra-statement consecutivity schedule constraints. In particular, if either of the two intra-statement consecutivity schedule constraints has failed, then the inter-statement consecutivity schedule constraint is considered to have failed as well. Furthermore, additional constraints are only added when the number h_i of schedule rows made equal to H_i (plus some linear combination) is still zero for both intra-statement consecutivity schedule constraints. In particular, using the notation of Definition 10 on page 14, the schedule distance of the new schedule row for all elements in R is set to be either 0 or 1, in correspondence with the inter-statement consecutivity schedule constraint satisfaction condition (29). That is, as long as either of the intra-statement consecutivity schedule constraints is still in the process of finding linear combinations of G_i, the distance is set to 0,

$$\forall \mathbf{a} \to \mathbf{b} \in R : f(\mathbf{b}) - f(\mathbf{a}) = 0, \tag{83}$$

while as soon as both have completed the linear schedule to cover G_i, the distance is set to 1,

$$\forall \mathbf{a} \to \mathbf{b} \in R : f(\mathbf{b}) - f(\mathbf{a}) = 1. \tag{84}$$

In accordance with (29), this distance-1 constraint is only applied for a single schedule row. The inter-statement consecutivity schedule constraint is ignored for any subsequent schedule rows.

3.3.3 Scheduling Algorithms

As explained in more detail by Verdoolaege and Janssens (2017, Section 6.2), the `isl` scheduler is combines two scheduling algorithms,

- the Feautrier scheduler (Feautrier 1992b), and
- (a variant of) the Pluto scheduler (Bondhugula, Baskaran, et al. 2008),

where the Pluto scheduler is used by default and the Feautrier scheduler is only used when the Pluto scheduler fails to produce a solution (because of extra constraints that are not imposed by the Feautrier scheduler). In the current implementation, intra-statement consecutivity schedule constraints and inter-statement consecutivity schedule constraints are only taken into account by the Pluto scheduler. This is similar to how proximity schedule constraints and coincidence schedule constraints are only taken into account by the Pluto scheduler. Whenever the Feautrier scheduler is used, it is only used to construct a single schedule row. This schedule row may violate some intra-statement consecutivity schedule constraints and/or inter-statement consecutivity schedule constraints. The scheduler continues with those constraints that can still be satisfied.

```
for (t = 0; t <= _PB_TSTEPS - 1; t++)
  for (i = 1; i<= _PB_N - 2; i++)
    for (j = 1; j <= _PB_N - 2; j++)
S:     A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                  + A[i][j-1] + A[i][j] + A[i][j+1]
                  + A[i+1][j-1] + A[i+1][j]
                  + A[i+1][j+1])/SCALAR_VAL(9.0);
```

Listing 12: Excerpt from `PolyBench/C 4.1 seidel-2d` benchmark, with an additional statement label and minor reformatting

Example 25. *Consider the code fragment shown in Listing 12. When run with the options* `--target=c --openmp --isl-schedule-outer-coincidence --consecutive-arrays='(A)'`*, the* `isl` *scheduler is forced to produce tilable bands with coincident members. It is impossible to produce such a band at the outer level and the Feautrier scheduler is therefore invoked to produce a single schedule row, resulting in*

$$\mathtt{S}[t,i,j] \to [4t+2i+j]. \tag{85}$$

In the absence of any intra-statement consecutivity schedule constraints, this schedule would get extended to

$$\mathtt{S}[t,i,j] \to [4t+2i+j,t,i], \tag{86}$$

resulting in the innermost transformed loop iterator appearing in the outermost index expression and therefore not achieving consecutivity to any extent. Taking into account intra-statement consecutivity schedule constraints leads to a better schedule. The (single) intra-statement consecutivity schedule constraint is

$$F = \left[\begin{array}{c} G \\ \hline H \end{array}\right] = \left[\begin{array}{ccc} 0 & 1 & 0 \\ \hline 0 & 0 & 1 \end{array}\right]. \tag{87}$$

Since

$$r_2 = \operatorname{rank}\begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} = 2 \qquad \textit{and} \qquad \operatorname{rank}\begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = 3, \tag{88}$$

while $f = 1$ and $h = 0$ at this point, failure (78) *has not been reached and the intra-statement consecutivity schedule constraint can continue to be taken into account. The next row needs to be a linear combination of G (and the outer row). The scheduler picks the most natural choice i for this row. The final row needs to have coefficients of the form*

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{89}$$

Natural choices would be j, i.e., $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ or $-4t$, i.e., $\begin{bmatrix} -4 & 0 & 0 \end{bmatrix}$. The scheduler picks j in this case because the sum of the coefficients (in absolute value) is smaller. The final (flattened) schedule is

$$\mathtt{S}[t,i,j] \to [4t+2i+j,i,j]. \tag{90}$$

Note that this schedule does not achieve array consecutivity because the schedule matrix has a determinant of 4. This is also evident from the resulting code in Listing 13 on the following page. Picking $-4t$ would obviously lead to a strided schedule dimension with the same factor 4. Picking $-t$ would not help either, because in that case the factor 4 would simply move into the index expression.

```
for (int c0 = 3; c0 < 4 * tsteps + 3 * n - 9; c0 += 1)
  #pragma omp parallel for
  for (int c1 =
       ppcg_max(1, -2 * tsteps - n + ppcg_fdiv_q(n + c0 + 1, 2) + 3);
       c1 < ppcg_min(n - 1, (c0 + 1) / 2); c1 += 1)
    for (int c2 = ppcg_max(-4 * tsteps + c0 - 2 * c1 + 4,
                           ((c0 - 2 * c1 - 1) % 4) + 1);
         c2 <= ppcg_min(n - 2, c0 - 2 * c1); c2 += 4)
      A[c1][c2] = (((((((((A[c1 - 1][c2 - 1] + A[c1 - 1][c2]) +
            A[c1 - 1][c2 + 1]) + A[c1][c2 - 1]) +
            A[c1][c2]) + A[c1][c2 + 1]) + A[c1 + 1][c2 - 1]) +
            A[c1 + 1][c2]) + A[c1 + 1][c2 + 1]) / 9.);
```

Listing 13: Transformed code for the input in Listing 12 on the previous page

3.3.4 Incremental Scheduling

The `isl` implementation of the Pluto scheduler constructs the schedule incrementally by default (Verdoolaege and Janssens 2017, Section 7.3). In particular, if there is more than one strongly connected component in the statement-level schedule constraint graph, then a schedule is first constructed for each component separately, after which the components are combined incrementally by scheduling them with respect to each other.

If any of the components (partially) satisfies some intra-statement consecutivity schedule constraints, then the scheduler needs to take care not to violate these intra-statement consecutivity schedule constraints when combining the components. In particular, if $h > 0$ for some statement in a component, i.e., if at least one schedule row has been set equal to a row in H, then an intra-statement consecutivity schedule constraint is introduced on the component that ensures that this row and all subsequent rows are unaffected (apart from possibly mixing in earlier rows). Let p be the position of the schedule row in the component schedule that corresponds to the first row of H. If this row belongs to an outer band, then set $p = 0$. Let v be the dimension of the component schedule. Then an intra-statement consecutivity schedule constraint is introduced with identity $F = I$ and f set to $v - p$.

Inter-statement consecutivity schedule constraints are less straightforward to carry over. In particular, in the case of intra-statement consecutivity schedule constraints, only those intra-statement consecutivity schedule constraints that have been satisfied inside the individual components need to be taken into account, but in the case of inter-statement consecutivity schedule constraints, there may be some that connect separate components and that would therefore still need to be imposed, rather than simply being preserved. Since there is no direct correspondence between the original intra-statement consecutivity schedule constraints and those introduced on the components. it is not clear how the intra-statement consecutivity schedule constraints referenced by an inter-statement consecutivity schedule constraint should be interpreted. Any inter-statement consecutivity schedule constraint is therefore taken to force the statements involved to belong to the same component. This ensures that there are no inter-statement consecutivity schedule constraints across components, but it does reduce the advantage of the incremental scheduling. No special treatment is required to preserve the satisfied inter-statement consecutivity schedule constraints inside the resulting components since the relative positions are not modified and the referenced intra-statement consecutivity schedule constraints are already preserved, ensuring that a distance-1 direction does not get mixed in with distance-0 directions.

Example 26. *Consider the code in Listing 14 on the following page. The*

```
void matmul(int N, int M, int K,
    __pencil_consecutive float A[restrict static N][K],
    __pencil_consecutive float B[restrict static K][M],
    __pencil_consecutive float C[restrict static N][M])
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < M; ++j) {
S:          C[i][j] = 0;
            for (int k = 0; k < K; ++k)
T:              C[i][j] += A[i][k] * B[k][j];
        }
}
```

Listing 14: Input file matmul2.c

intra-statement consecutivity schedule constraints for statement `T` *are the same as those* (51) *derived in Example 19 on page 20. For* `S`*, the intra-statement consecutivity schedule constraint is*

$$F_{\mathtt{S}} = \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right]. \tag{91}$$

There are no inter-statement consecutivity schedule constraints because the accesses to the next memory element always come from the same statement. The two statements are therefore first scheduled individually, resulting in the schedules

$$\{\, \mathtt{S}[i,j] \to \mathtt{C0}[i,j] \,\} \qquad \textit{and} \qquad \{\, \mathtt{T}[i,j,k] \to \mathtt{C1}[i,k,j] \,\}. \tag{92}$$

In both cases, the second schedule dimension corresponds to the first row of the respective H matrices. The intra-statement consecutivity schedule constraints on the clusters `C0` *and* `C1` *are therefore*

```
{ C0[a, b] -> [[(a)] -> [(b)]] }
{ C1[a, b, c] -> [[(a)] -> [(b), (c)]] }
```

The proximity schedule constraints between the clusters are

```
[M, N, K] -> {
    C1[a, b, c] -> C1[a, 1 + b, c] :
        0 <= a < N and 0 <= b <= -2 + K and 0 <= c < M;
    C0[a, b] -> C1[a, 0, b] :
        K > 0 and 0 <= a < N and 0 <= b < M }
```

These result in the cluster schedule

$$\{\, \mathtt{C0}[a,b] \to [a,0,b]; \mathtt{C1}[a,b,c] \to [a,b,c] \,\}, \tag{93}$$

satisfying the cluster intra-statement consecutivity schedule constraints. Combined with the individual schedules (92)*, the final result is*

$$\{\, \mathtt{S}[i,j] \to [i,0,j]; \mathtt{T}[i,j,k] \to [i,k,j] \,\}. \tag{94}$$

3.4 Solving Consecutivity Constraints

This section describes how the constraints on schedule coefficients derived from the consecutivity schedule constraints are solved by the ILP solver used by the `isl` scheduler. The mechanism for handling other constraints on schedule coefficients is first summarized and is then adjusted to handle those derived from intra-statement consecutivity schedule constraints and inter-statement consecutivity schedule constraints.

3.4.1 The `isl` scheduler ILP problem

As explained in detail by Verdoolaege and Janssens (2017, Section 6.5), `isl` computes schedules by solving ILP problems in the schedule coefficients. Generally speaking, there are two classes of constraints that need to be imposed. The first class consists of those derived from validity, coincidence and proximity schedule constraints. These are of the form

$$\forall \mathbf{a} \to \mathbf{b} \in R : f(\mathbf{b}) - f(\mathbf{a}) \geq 0 \tag{95}$$

or

$$\forall \mathbf{a} \to \mathbf{b} \in R : f(\mathbf{b}) - f(\mathbf{a}) \leq u(\mathbf{n}), \tag{96}$$

with $\mathbf{n}$ the symbolic constants. These are converted to constraints on the schedule coefficients $\mathbf{c}$ by applying the Farkas lemma (Schrijver 1986, Corollary 7.1h, page 93; Feautrier 1992a, Theorem 7).

The second class is formed by the linear independence constraints, which require the next schedule row $\mathbf{c}$ of some statement to be linearly independent of the previous schedule rows T_0, i.e.,

$$\neg \exists \mathbf{y} : \mathbf{c}^{\mathrm{t}} = \mathbf{y}^{\mathrm{t}} T_0. \tag{97}$$

These constraints are imposed by computing (a basis for) the orthogonal complement U of T_0, i.e.,

$$T_0 \, U^{\mathrm{t}} = 0 \qquad \text{and} \qquad \operatorname{rank} T_0 + \operatorname{rank} U = n, \tag{98}$$

and requiring

$$U\mathbf{c} \neq \mathbf{0}. \tag{99}$$

Note that the final paragraph of Verdoolaege and Janssens (2017, Section 6.5.6) mistakenly suggests that no orthogonal complement is being computed. The requirement (99) itself is imposed using a backtracking search that imposes

$$U_i \mathbf{c} \geq 1 \qquad \text{or} \qquad U_i \mathbf{c} \leq -1 \tag{100}$$

for each row U_i of U in turn until a solution is found. Note that these constraints are only imposed if the requirement (99) is not already satisfied. In practice, when a schedule is computed incrementally, such constraints typically only need to be imposed on a single statement since as soon as the schedule coefficients of one statement are linearly independent of previous rows, the constraints that connect statements will usually force the schedule coefficients of other statements to become linearly independent of previous rows as well. The rows of U are also normalized to favor schedules with zero values for later schedule coefficients and a positive value for the first schedule coefficient involved.

The constraints from the first class are directly encoded in the ILP problem sent to the ILP solver. If there are any coincidence schedule constraints, then the ILP solver is potentially run twice, once with the coincidence schedule constraints included and, if this does not produce a solution, then once more without the coincidence schedule constraints. As soon as the coincidence schedule constraints prevent the ILP solver from finding a solution, they are dropped from consideration throughout the construction of the band. The constraints from the second class are passed separately as constraints on "*regions*" of schedule coefficients (in practice, those corresponding to a particular statement). Any region with violated constraints results in constraints being added to the ILP problem (and possibly removed again) during the backtracking search.

Once a solution has been found, backtracking continues, but the search is narrowed to "significantly better" solutions. In practice, this means that a

solution with a parametric bound on the distances over proximity schedule constraints may be replaced by one with a non-parametric bound and that a solution with a non-zero bound may be replaced by one with a zero bound.

As described below, the intra-statement consecutivity schedule constraints and inter-statement consecutivity schedule constraints are handled by the same backtracking procedure (with some modifications). This means that, just like the proximity schedule constraints, they are considered to be of lower priority than the coincidence schedule constraints since no ILP problem without coincidence schedule constraints will be constructed if the ILP problem with coincidence schedule constraints produces a solution.

3.4.2 Intra-statement Consecutivity Schedule Constraints

The constraints on the schedule coefficients derived from the intra-statement consecutivity schedule constraints and summarized in Table 11 on page 29 are similar to the second class of constraints (97) already used in the `isl` scheduler. In particular, there are three types of elementary constraints,

1. linear independence

   This is of the same form as (97) and is handled in the same way by computing the orthogonal complement and forcing the new schedule row to not be orthogonal to this orthogonal complement. Note that compared to the linear independence constraints of Verdoolaege and Janssens (2017, Section 6.5.6) that simply enforce the next schedule row to be linearly independent of the schedule computed so far (i.e., T_0), the linear independence constraints in Table 11 have more rows with respect to which the next row needs to be linearly independent. This means that there are fewer rows in the orthogonal complement and therefore fewer cases to handle.

2. linear combination

   This is the opposite of linear independence. The same orthogonal complement U is computed, but now the new schedule row is forced to be orthogonal to this orthogonal complement, i.e.,

   $$U\mathbf{c} = \mathbf{0}. \tag{101}$$

   Note that these are linear constraints, meaning that they can be posted as a whole without any backtracking.

3. specialized linear combination

   This is a constraint of the form

   $$\mathbf{c}^{\mathrm{t}} = \mathbf{a}^{\mathrm{t}} + \mathbf{b}^{\mathrm{t}} M, \tag{102}$$

   where $\mathbf{a}$ is linearly independent of M. Let U be the orthogonal complement of

   $$\begin{bmatrix} M \\ \mathbf{a}^{\mathrm{t}} \end{bmatrix}. \tag{103}$$

   Then

   $$U\mathbf{c} = \mathbf{0}. \tag{104}$$

   Moreover, if U' is the orthogonal complement of M, then the condition (102) also implies

   $$U'\mathbf{c} = U'\mathbf{a}. \tag{105}$$

Because the conditions (104) also hold, the matrix U' in this last condition can be replaced by $U'' = U' \setminus U$. The combined constraint on the schedule coefficients $\mathbf{c}$ is therefore

$$\begin{bmatrix} U \\ \bar{U''} \end{bmatrix} \mathbf{c} = \begin{bmatrix} \mathbf{0} \\ \bar{U''}\bar{\mathbf{a}} \end{bmatrix}. \tag{106}$$

The new types of constraints both force some linear combinations of the schedule coefficients to have a fixed value. When a (specialized) linear combination is combined with a linear independence in a conjunction, then constraints of the form

$$U\mathbf{c} = \mathbf{0} \qquad \text{and} \qquad V\mathbf{c} \neq \mathbf{0} \tag{107}$$

need to be satisfied together. In this case, V can again be replaced by $V' = V \setminus U$. It is the rows of this V' that are normalized to favor schedules with zero values for later schedule coefficients and a positive value for the first schedule coefficient involved.

Example 27. *Continuing from Example 25 on page 31, during the computation of the second schedule row, this second row needs to be a linear combination of*

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{108}$$

and linearly independent of

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}. \tag{109}$$

This means

$$\begin{bmatrix} 1 & 0 & -4 \end{bmatrix} \mathbf{c} = \mathbf{0} \qquad \textit{and} \qquad \begin{bmatrix} 1 & -2 & 0 \end{bmatrix} \mathbf{c} \neq \mathbf{0}. \tag{110}$$

The computation

$$\begin{bmatrix} 1 & -2 & 0 \end{bmatrix} \setminus \begin{bmatrix} 1 & 0 & -4 \end{bmatrix} \tag{111}$$

yields

$$\begin{bmatrix} 0 & -1 & 2 \end{bmatrix}, \tag{112}$$

which is normalized to

$$\begin{bmatrix} 0 & 1 & -2 \end{bmatrix}. \tag{113}$$

As already mentiond in Example 25 on page 31, this results in second schedule row i.

The final row then needs to be equal to

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \tag{114}$$

while also being linearly independent of

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{115}$$

The orthogonal complement of

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{116}$$

is empty, while the orthogonal complement of

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{117}$$

is

$$\begin{bmatrix} 1 & 0 & -4 \end{bmatrix}. \tag{118}$$

This yields the constraints

$$\begin{bmatrix} 1 & 0 & -4 \end{bmatrix} \mathbf{c} = \begin{bmatrix} 1 & 0 & -4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -4 \end{bmatrix} \tag{119}$$

and

$$\begin{bmatrix} 1 & 0 & -4 \end{bmatrix} \mathbf{c} \neq \mathbf{0}, \tag{120}$$

where the non-zero constraint is redundant here because the same linear combination is forced to be equal to -4*. As explained in Example 25 on page 31, these constraints cause the scheduler to pick final schedule row* i*.*

As in the case of the standard linear independence constraints, a constraint on the schedule coefficients is only enforced if it is not already satisfied by the current ILP solution. However, the backtracking search needs to be modified in several ways.

- In contrast to the standard linear independence constraints that are required to produce a schedule with linearly independent rows, the constraints on schedule coefficients derived from intra-statement consecutivity schedule constraints are *optional*. That is, a schedule not satisfying some intra-statement consecutivity schedule constraint is still a valid, if suboptimal, schedule. Besides the $2n$ cases of the form (100), the search therefore also needs to consider the case where the constraint is not imposed, but is *disabled* instead. The constraint needs to be disabled to avoid the constraint being considered at nested levels in the search. It is re-enabled when backtracking out of the level that disabled the constraint. Optional constraints are considered before required constraints, i.e., those that are required for linear independence of the schedule. Note that the optional constraints that involve some linear independence subsume the required linear independence constraints on the same statement. That is, when this part of the optional constraint is being enforced, the corresponding required constraint will not be triggered.

- The new types of constraints have a *fixed* part that is enforced in all the linear independence cases (100), but not in the case where the constraint is disabled. If the constraint does not involve a linear independence part, then there are two states, one where the fixed part is enforced and one where the constraint is disabled.

- The constraint may be *disjunctive*, in which case it is only triggered when all of the disjuncts are violated by the current ILP solution. When it is triggered, the first disjunct that has not been disabled at previous levels of the backtracking search is enforced. If this does not result in a solution, then the disjunct will be disabled and the next disjunct will be enforced until all disjuncts have been considered.

- Finally, a (possibly disjunctive) constraint may be *conditional* on the previous (possibly disjunctive) constraint. In this case, the entire disjunctive constraint is ignored until the final disjunct of the previous constraint has been disabled.

```
void matmul(int N, int M, int K, float alpha, float beta,
    __pencil_consecutive float A[restrict static K][N],
    __pencil_consecutive float B[restrict static K][M],
    __pencil_consecutive float C[restrict static N][M])
{
    for (int i = 0; i < N; ++i) {
        float c[M];
        for (int j = 0; j < M; ++j)
S:          c[j] = -0.0f;
        for (int j = 0; j < M; ++j) {
            for (int k = 0; k < K; ++k) {
T:              c[j] += A[k][i] * B[k][j];
            }
U:          C[i][j] = C[i][j] * beta + alpha * c[j];
        }
    }
}
```

Listing 15: Input file matmul3.c

Whenever a solution has been found, all optional constraints satisfied by the solution are turned into required constraints, ensuring that any improved solution has at least the same satisfied intra-statement consecutivity schedule constraints. This could be further refined to enforce that the *number* of satisfied intra-statement consecutivity schedule constraints does not decrease. Note, in particular, that the backtracking search is currently not continued for the purpose of increasing the number of satisfied intra-statement consecutivity schedule constraints, but only for obtaining a "significantly better" solution, as described in Section 3.4.1.

While this extended search procedure appears to work reasonably well in those cases where intra-statement consecutivity schedule constraints end up getting satisfied, it can in some cases result in needless transformations when it turns out that they cannot be satisfied in the end.

Example 28. *Consider the code in Listing 15, which represents a form of matrix multiplication with a local buffer. In principle, a schedule that interchanges the* `j`*-loop and the* `k`*-loop can be found, which would at least make the accesses to* `B` *consecutive. However, independently of any consecutivity constraints, the* `isl` *scheduler is unable to find such a schedule, because it optimizes for* outer *permutability. In particular, it will favor a schedule with a two-dimensional outer permutable band, but the schedule function k cannot be part of such an outer permutable band.*[2] *The current approach for constructing intra-statement consecutivity schedule constraints in* `PPCG`*, as described in Section 3.2.1, may construct multiple intra-statement consecutivity schedule constraints for each statement. For statement* `T`*, the list contains*

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \tag{121}$$

in the first positions, with their relative order depending on the order in which the accesses are considered. Both these intra-statement consecutivity schedule con-

[2] It would be possible to favor inner permutability by cutting a non-innermost band after the outer element and computing a nested band within the cut-off band, but this is beyond the scope of this report.

straints force the `k`*-loop outermost, which is impossible. These intra-statement consecutivity schedule constraints therefore fail at the outermost level and are ignored from then on. The next elements in the list of intra-statement consecutivity schedule constraints only cover individual references, i.e.,*

$$\left[\begin{array}{ccc} 0 & 0 & 1 \\ 1 & 0 & 0 \end{array}\right] \qquad \textit{and} \qquad \left[\begin{array}{ccc} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right]. \tag{122}$$

There are two ways of satisfying either of these intra-statement consecutivity schedule constraints at the outer level, the outer schedule row should be a linear combination of k (still impossible), or it should be linearly independent of either i or j. Either of these succeeds at the outer level with schedule row j or i. Each of these choices invalidates the other intra-statement consecutivity schedule constraint. At the second level, the schedule row should be a linear combination of k and either j or i, which is impossible without breaking the band. The remaining intra-statement consecutivity schedule constraint therefore fails at the second level. The result then depends on the order of the intra-statement consecutivity schedule constraints. If schedule row j gets selected at the outer level, then this will prevent consecutivity on `C` *as well. Note that the solver will always consider the (first) constraint on statement* `T` *before the constraint on statement* `U` *because the latter is lower-dimensional and the initial trivial solution satisfies the linear dependence disjunct. This means that the first violated (disjunctive) constraint will always be one on* `T`*.*

An alternative to considering the intra-statement consecutivity schedule constraints (on `T`*) in order would be to try and cover multiple intra-statement consecutivity schedule constraints for the same statement together, hoping that this would increase the chance that at least one of them survives. However, this may actually lead to more complicated schedules without any added benefit. In this case, there are again two ways of satisfying both intra-statement consecutivity schedule constraints at the outer level, the outer schedule row should be a linear combination of k, or it should be linearly independent of both i and j (individually). The scheduler would therefore pick the outer schedule row $i + j$. At the second level, the schedule row should should be a linear combination of k and $i + j$, which is again impossible without breaking the band. Both intra-statement consecutivity schedule constraints would therefore fail at the second level. The final schedule (restricted to* `T`*) would then be*

$$\{\, \mathtt{T}[i, j, k] \to [i + j, i, k] \,\}, \tag{123}$$

which is no better than the original schedule.

3.4.3 Inter-statement Consecutivity Schedule Constraints

The constraints derived from the inter-statement consecutivity schedule constraints set the schedule distance to 0 (83) or 1 (84). These constraints belong to the first class of constraints that need to be imposed on the ILP problem (95)(96). They could then also be converted to constraints on schedule coefficients through an application of the Farkas lemma by considering the equality constraints as pairs of inequality constraints. However, since only equality constraints are involved, it is much simpler to write the equality constraint that needs to be enforced as a linear combination of the equality constraints satisfied by R. In particular, let

$$\{\, \mathbf{x} \to \mathbf{y} : A\mathbf{x} + B\mathbf{y} + M\mathbf{n} + \mathbf{d} = 0 \,\} \tag{124}$$

be the affine hull of R. Then an equality constraint

$$\mathbf{c}^{\mathrm{x}} \cdot \mathbf{x} + \mathbf{c}^{\mathrm{y}} \cdot \mathbf{y} + \mathbf{c}^{\mathrm{n}} \cdot \mathbf{n} + c^{\mathrm{c}} = 0 \tag{125}$$

is valid for all elements in R iff it can be written as

$$\mathbf{c}^{\mathrm{x}} \cdot \mathbf{x} + \mathbf{c}^{\mathrm{y}} \cdot \mathbf{y} + \mathbf{c}^{\mathrm{n}} \cdot \mathbf{n} + c^{\mathrm{c}} = \boldsymbol{\lambda}^{\mathrm{t}} \left(A\mathbf{x} + B\mathbf{y} + M\mathbf{n} + \mathbf{d} \right) \tag{126}$$

for some $\boldsymbol{\lambda}$. That is,

$$Q \begin{bmatrix} \mathbf{c}^{\mathrm{x}} \\ \mathbf{c}^{\mathrm{y}} \\ \mathbf{c}^{\mathrm{n}} \\ c^{\mathrm{c}} \end{bmatrix} = 0 \tag{127}$$

with Q the orthogonal complement of

$$\begin{bmatrix} A & B & M & \mathbf{d} \end{bmatrix}. \tag{128}$$

In the special case of the constraint

$$f_k(\mathbf{y}) - f_j(\mathbf{x}) = v, \tag{129}$$

with v either 0 or 1 and

$$f_j(\mathbf{x}) = \mathbf{c}_j^{\mathrm{x}} \cdot \mathbf{x} + \mathbf{c}_j^{\mathrm{n}} \cdot \mathbf{n} + c_j^{\mathrm{c}}, \tag{130}$$

this specializes to

$$Q \begin{bmatrix} -\mathbf{c}_j^{\mathrm{x}} \\ \mathbf{c}_k^{\mathrm{y}} \\ \mathbf{c}_k^{\mathrm{n}} - \mathbf{c}_j^{\mathrm{n}} \\ c_k^{\mathrm{c}} - c_j^{\mathrm{c}} - v \end{bmatrix} = 0. \tag{131}$$

That is,

$$Q \begin{bmatrix} -\mathbf{c}_j^{\mathrm{x}} \\ \mathbf{c}_k^{\mathrm{y}} \\ \mathbf{c}_k^{\mathrm{n}} - \mathbf{c}_j^{\mathrm{n}} \\ c_k^{\mathrm{c}} - c_j^{\mathrm{c}} \end{bmatrix} = 0 \tag{132}$$

for $v = 0$ and

$$Q \begin{bmatrix} -\mathbf{c}_j^{\mathrm{x}} \\ \mathbf{c}_k^{\mathrm{y}} \\ \mathbf{c}_k^{\mathrm{n}} - \mathbf{c}_j^{\mathrm{n}} \\ c_k^{\mathrm{c}} - c_j^{\mathrm{c}} \end{bmatrix} = Q \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ 1 \end{bmatrix} \tag{133}$$

for $v = 1$. Both constraint (132) and constraint (133) have the form of a "region" constraint with a fixed value. However, the region does not correspond to the variable coefficients of a single statement in this case, but instead on all coefficients of a pair of statements. In practice, the region is taken to cover all coefficients for simplicity.

Just like any other optional region constraint with a fixed value and no linear independence, the region constraint can be in two states after it has been activated: the fixed value is enforced, or the region has been disabled. In principle the fixed value derived from an inter-statement consecutivity schedule constraint should only be enforced if the region constraints of the corresponding intra-statement consecutivity schedule constraints are satisfied. The current implementation does not explicitly check for this condition since it is fairly unlikely that the inter-statement consecutivity schedule constraint can be imposed if the corresponding intra-statement consecutivity schedule constraints cannot be satisfied.

4 Partial Rescheduling

Just like any other form of schedule constraints, the consecutivity schedule constraints described above can be added to the constraints used for scheduling the entire program fragment under consideration. The consecutivity schedule constraints may however conflict with the proximity schedule constraints, causing statement instances to be scheduled further apart then they would have been without the consecutivity schedule constraints. In particular, this can result in higher memory requirements (after memory requirement optimization, Darte et al. 2005). It may therefore in some cases be useful to only apply the consecutivity schedule constraints at a later stage, e.g., inside the innermost tiles. Since the output of the `isl` scheduler only marks permutable bands and leaves it up to the caller to decide which of these band to tile and by how much, it cannot make a distinction between the schedule that is used for tiling and the schedule that would be used inside the tile. One way of taking into account the consecutivity schedule constraints in this tile schedule is then to *reschedule* the part of the schedule tree that corresponds to the tile after tiling. In the `PPCG` implementation, this rescheduling can be enabled using the `--consecutivity-level=intra-tile` command line option. In particular, the (default) `--consecutivity-level=global` option tells `PPCG` to take consecutivity schedule constraints into account at the global level, while the `--consecutivity-level=intra-tile` option tells `PPCG` to ignore consecutivity schedule constraints at the global level and to instead reschedule the innermost tiled bands taking into account the consecutivity schedule constraints. This rescheduling is similar to the intra-tile interchanges suggested by Bondhugula, Hartono, et al. (2008) (see Section 2.8), except that the intra-tile schedule is completely recomputed, rather than being restriced to interchange on the already computed schedule.

Example 29. *Continuing from Example 15 on page 17, consider once more the code in Listing 4 on page 17. When consecutivity schedule constraints are applied at the global level, the resulting code is as shown in Listing 5 on page 17. This code has consecutivity in the accesses to both the* `A` *and* `C` *arrays, but requires an entire copy to be stored in* `tmp` *in between the two loop nests. When consecutivity schedule constraints are not taken into account at the global level, the generated schedule corresponds to the original execution order (or an interchange of the two loops, depending on the order in which the consecutive arrays are considered). This schedule is shown in Listing 16 on the following page. After tiling, the schedule is as shown in Listing 17 on the next page, with the pointer (*`# YOU ARE HERE`*) at the intra-tile schedule. Rescheduling this subtree, taking into account consecutivity schedule constraints, results in the schedule tree shown in Listing 18 on the following page. The final code is shown in Listing 19 on page 43. Note that this code also has consecutivity in the accesses to both the* `A` *and* `C` *arrays and that now only enough room for a tile needs to be allocated for* `tmp`*. (The memory requirement optimization is not shown in Listing 19.)*

Rescheduling a subtree of a schedule tree essentially boils down to scheduling the statement instances active at the top of the subtree and grafting the resulting schedule tree at the position of the original subtree. However, some linear combinations of statement indices may already be fixed by outer schedule bands at this position. The generated schedule at this subtree should then not contain any schedule rows that are linearly dependent on these linear combinations. This is accomplished by extracting the linear combinations from the prefix schedule (the concatenation of all outer band schedules) and then treating these linear combinations in the same way as earlier schedule rows. In particular, they are

```
domain: "[N] -> { S[i, j] : 0 <= i < N and 0 <= j < N;
        T[i, j] : 0 <= i < N and 0 <= j < N }"
child:
  schedule: "[N] -> [{ S[i, j] -> [(i)]; T[i, j] -> [(i)] },
                    { S[i, j] -> [(j)]; T[i, j] -> [(j)] }]"
  permutable: 1
  coincident: [ 1, 1 ]
  child:
    sequence:
    - filter: "[N] -> { S[i, j] }"
    - filter: "[N] -> { T[i, j] }"
```

Listing 16: Original schedule for the code in Listing 5 on page 17

```
domain: "[N] -> { S[i, j] : 0 <= i < N and 0 <= j < N;
        T[i, j] : 0 <= i < N and 0 <= j < N }"
child:
  schedule: "[N] -> [{ S[i, j] -> [(32*floor((i)/32))];
                       T[i, j] -> [(32*floor((i)/32))] },
                    { S[i, j] -> [(32*floor((j)/32))];
                       T[i, j] -> [(32*floor((j)/32))] }]"
  permutable: 1
  coincident: [ 1, 1 ]
  options: "{ atomic[i0] : 0 <= i0 <= 1 }"
  child:
    # YOU ARE HERE
    schedule: "[N] -> [{ S[i, j] -> [(i - 32*floor((i)/32))];
                       T[i, j] -> [(i - 32*floor((i)/32))] },
                    { S[i, j] -> [(j - 32*floor((j)/32))];
                       T[i, j] -> [(j - 32*floor((j)/32))] }]"
    permutable: 1
    coincident: [ 1, 1 ]
    child:
      sequence:
      - filter: "[N] -> { S[i, j] }"
      - filter: "[N] -> { T[i, j] }"
```

Listing 17: Tiled schedule for the code in Listing 5 on page 17

```
domain: "[N] -> { S[i, j] : 0 <= i < N and 0 <= j < N;
        T[i, j] : 0 <= i < N and 0 <= j < N }"
child:
  schedule: "[N] -> [{ S[i, j] -> [(32*floor((i)/32))];
                       T[i, j] -> [(32*floor((i)/32))] },
                    { S[i, j] -> [(32*floor((j)/32))];
                       T[i, j] -> [(32*floor((j)/32))] }]"
  permutable: 1
  coincident: [ 1, 1 ]
  options: "{ atomic[i0] : 0 <= i0 <= 1 }"
  child:
      sequence:
      - filter: "[N] -> { S[i, j] }"
        child:
          schedule: "[N] -> [{ S[i, j] -> [(i)] }, { S[i, j] -> [(j)] }]"
          permutable: 1
          coincident: [ 1, 1 ]
      - filter: "[N] -> { T[i, j] }"
        child:
          schedule: "[N] -> [{ T[i, j] -> [(j)] }, { T[i, j] -> [(i)] }]"
          permutable: 1
          coincident: [ 1, 1 ]
```

Listing 18: Result of rescheduling intra-tile schedule for the code in Listing 5 on page 17

```
float tmp[N][N];
for (int c0 = 0; c0 < N; c0 += 32)
  for (int c1 = 0; c1 < N; c1 += 32) {
    for (int c2 = c0; c2 <= min(N - 1, c0 + 31); c2 += 1)
      for (int c3 = c1; c3 <= min(N - 1, c1 + 31); c3 += 1)
        tmp[c2][c3] = A[c2][c3];
    for (int c2 = c1; c2 <= min(N - 1, c1 + 31); c2 += 1)
      for (int c3 = c0; c3 <= min(N - 1, c0 + 31); c3 += 1)
        C[c2][c3] = tmp[c3][c2];
  }
```

Listing 19: The code in Listing 4 on page 17 transformed using the schedule in Listing 18 on the previous page

used to determine whether a newly compute schedule row is linearly independent and to determine the state during the handling of consecutivity schedule constraints. The entire prefix schedule (not just the extracted linear part) is also used to filter out all schedule constraints between statement instances that are not coscheduled by this prefix schedule.

The extraction of the linear part needs to take into account any valid prefix schedule and not just those that may appear in schedule trees generated by the `isl` scheduler. In particular, it needs to take into account quasi-affine expressions and piecewise expressions. It is impossible to extract all valid linear combinations of quasi-affine expressions that are purely affine, but some frequently occurring cases can be handled relatively easily. In particular, if the prefix schedule contains both $\lfloor i/N \rfloor$ and $i \bmod N$ for some fixed value N, then $i = N(\lfloor i/N \rfloor) + (i \bmod N)$ is also fixed by the prefix schedule. This may happen, for example, inside the point band of some outer tiled band. Inside `isl`, all quasi-affine expressions are formulated in terms of floor-expressions. That is, $i \bmod N$ is represented as

$$i \bmod N = i - N \lfloor i/N \rfloor . \tag{134}$$

Finding purely affine linear combinations is then a matter of using schedule rows that involve a floor-expression to eliminate this floor-expression from other schedule rows. The remaining schedule rows that do not involve any floor-expressions are then taken as the linear part of the prefix schedule.

If the prefix schedule is piecewise, then different linear combinations may get fixed on different parts of the domain. The objective here is to find those linear combinations that are fixed on all parts of the domain. These are obtained by collecting all directions that cause a linear combination *not* to be fixed (the orthogonal complement of the fixed directions) over all parts of the domain and taking the orthogonal complement of those. In polyhedral terminology, this means that the convex hull of the fixed parts is obtained as the dual of the intersection of the duals.

Example 30. *Consider the piecewise prefix schedule*

$$\{ \mathtt{S}[i,j,k] \to [i, i+j] : i \geq 0; \mathtt{S}[i,j,k] \to [i+k, k] : i < 0 \}. \tag{135}$$

The linear parts of the schedule on the different parts of the domain are

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}. \tag{136}$$

The corresponding orthogonal complements are

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, \tag{137}$$

which combine to

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{138}$$

The complement of this matrix in turn is

$$\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}. \tag{139}$$

That is, the linear combination i is fixed on both parts of the domain.

5 Matrix Operations

This section describes some basic operations on matrices and their implementation in `isl`. Note that these implementations may not necessarily be the most efficient implementations.

The core mechanism is the method of 方程 (Gaussian elimination), in particular in the form of the Hermite normal form H (Schrijver 1986, Chapter 4) of a matrix M,

$$H = M\,U, \tag{140}$$

where H is in column echelon form and U is a unimodular matrix. A matrix is in column echelon form if the first non-zero entry (if any) in a column is positive and if the position of the first non-zero entry increases with the column index, with the zero columns appearing last. The matrix H has additional properties that are not relevant for this section. With $Q = U^{-1}$, equation (140) can also be written

$$M = H\,Q. \tag{141}$$

5.1 Rank

The rank of a matrix M can be computed by determining the number of non-zero columns in the Hermite normal form of M.

5.2 Orthogonal Complement

A basis for the orthogonal complement of the space spanned by the rows of a matrix M can be obtained by computing its Hermite normal form and taking the last columns of the corresponding unimodular matrix U. These columns correspond to the zero columns of H and are therefore orthogonal to the rows of M.

5.3 Basis

A basis for the space spanned by the rows of a matrix M itself can be obtained by computing its Hermite normal form and taking the first $\operatorname{rank}(M)$ rows of the corresponding unimodular matrix $Q = U^{-1}$ (141). Since only the first $\operatorname{rank}(M)$ columns of H are non-zero, the rows of M are linear combinations of those rows of Q. Alternatively, Gaussian elimination can be applied directly to M. After removal of the zero rows (if any) from the result, a basis is obtained as well.

5.4 Basis Extension

The basis extension of a matrix A to cover B, written $B \setminus A$ is formed by rows that extend a basis of A to a basis that also covers B. Computing the Hermite normal form of the rows of A and B, the equation (141) can be written as

$$\begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} H_1 & 0 & 0 \\ H_2 & H_3 & 0 \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \end{bmatrix}, \tag{142}$$

with H_1 and H_3 of full column rank. The rows of Q_1 therefore form a basis for A, while Q_2 extends this basis to a basis that also covers B.

6 Discussion

There are different strategies that can be followed for trying to achieve some form of consecutivity that may each have its advantages and disadvantages. This report only explores one of these paths. The simplest approach may be to look for suitable interchange and reversal transformations on entire tilable loop nests, either on the polyhedral schedule or on the AST directly. Such an approach will miss opportunities where some skewing needs to be applied, but skewing can also have negative effects. It will also miss cases where different transformations need to be applied to different statements, such as the one in Example 29 on page 41.

Another approach is to try and integrate support for consecutivity in the scheduler. There are different ways of constructing a schedule. Within polyhedral compilation, a popular approach is to construct constraints on schedule coefficients through an application of the Farkas lemma, but there are also other approaches such as those based on transitive closures (e.g., Bielecki et al. 2017). Within the Farkas based approaches, there are two main groups, those that compute a schedule row by row and those that compute a schedule in one shot. In theory, the advantage of a one-shot schedule is that constraints can be imposed across schedule rows. However, in practice, it is not always easy to express such constraints. For example, it is not clear how to enforce linear independence of schedule rows unless the schedule is restricted to interchanges and reversals.

This report describes consecutivity support in a row-by-row Farkas based scheduler. It does *not*, however, add full support for vectorization. In fact, the current scheduler will try to place parallel schedule rows outermost, while consecutive rows are placed innermost, reducing the chance of finding a schedule row that is both parallel and consecutive. Moreover, the scheduler first looks for parallel rows. If this causes any consecutivity schedule constraints to fail, then these will simply be ignored. Proper support for vectorization would require the scheduler to either allow consecutivity schedule constraints to overrule a choice for an (outer) parallel row or to compute the innermost rows (that are both parallel and consecutive) first. Allowing consecutivity schedule constraints to take priority over coincidence schedule constraints could be achieved by not adding the schedule coefficient constraints corresponding to the latter directly to the ILP, but instead to add them as a single optional region after the regions corresponding to consecutivity schedule constraints.

As described in Section 3.4.2, once a solution has been found, the scheduler will currently only look for solutions that are (significantly) better on the proximity schedule constraints, ensuring only that this does not break any already handled intra-statement consecutivity schedule constraints. This means in particular, that the scheduler currently does not try to maximize the number of handled intra-statement consecutivity schedule constraints. If this number

should be optimized then the backtracking search should keep track of the number and use it as a bound during the search.

An alternative to the introducion of the consecutivity schedule constraints of Section 3.1 would be to use proximity schedule constraints to try and bring accesses to consecutive elements close to each other. This has the advantage of not having to introduce a separate mechanism and in particular of not having to distinguish between intra-statement consecutivity schedule constraints and inter-statement consecutivity schedule constraints. However, it is not clear how to combine several consecutivity constraints as in Section 3.2.1 if they are formulated in terms of proximity schedule constraints. The standard mechanism for handling proximity schedule constraints also has no mechanism for dropping constraints that have "failed" (when interpreted as targeting consecutivity) other than the removal of constraints carried by an entire band. In case of temporal reuse in an aray reference, a naive formulation in terms of proximity schedule constraints would force the temporal reuse innermost because there would be a dependence between each instance accessing an element and each instance accessing the next element, while the constraint on schedule coefficients of Section 3.3.1 and their solution in Section 3.4.2 also allow temporal reuse in outer positions. Furthermore, such large groups of proximity schedule constraints may result in infeasibility of the ILP problem as discussed by Verdoolaege and Janssens (2017, Section 6.6.3). When mixed in with other proximity schedule constraints, any such constraints that force the schedule distance to be different from zero at an outer level may render those directed at consecutivity ineffective. Finally, proximity schedule constraints are not directional. That is, they only bring statement instances close to each other, but do not ensure that one appears before the other. The approach of Zinenko et al. (2017) resolves some of these issues by introducing specialized spatial proximity schedule constraints.

Acknowledgments

This research was supported by Xilinx. The authors would also like to thank Uday Bondhugula for his feedback.

References

Anderson, Jennifer M., Saman P. Amarasinghe, and Monica S. Lam (1995). "Data and Computation Transformations for Multiprocessors". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA: ACM, pp. 166–178. DOI: 10.1145/209936.209954. [4]

Bastoul, Cédric and Paul Feautrier (Apr. 2003a). "Improving Data Locality by Chunking". In: *Compiler Construction: 12th International Conference (CC 2003)*. Ed. by Görel Hedin. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 320–334. DOI: 10.1007/3-540-36579-6_23. [3, 6]

Bastoul, Cédric and Paul Feautrier (Jan. 2003b). "Reordering methods for data locality improvement". In: *CPC'10 Compilers for Parallel Computers*. Amsterdam, The Netherlands, pp. 187–196. [5]

Bastoul, Cédric and Paul Feautrier (Aug. 2004). "More Legal Transformations for Locality". In: *Euro-Par'10 International Euro-Par conference*. Vol. 3149. Lecture Notes in Computer Science. Pisa, pp. 272–283. DOI: 10.1007/978-3-540-27866-5_36. [6, 27]

Bielecki, Wlodzimierz, Marek Palkowski, and Piotr Skotnicki (Oct. 2017). "Generation of parallel synchronization-free tiled code". In: *Computing*. DOI: 10.1007/s00607-017-0576-3. [45]

Bik, Aart J. C. (1996). "Compiler Support for Sparse Matrix Computations". PhD thesis. The Netherlands: University of Leiden. [5]

Bondhugula, Uday, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan (Apr. 2008). "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *International Conference on Compiler Construction (ETAPS CC)*. DOI: 10.1007/978-3-540-78791-4_9. [6, 30]

Bondhugula, Uday, Albert Hartono, J. Ramanujam, and P. Sadayappan (2008). "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '08. Tucson, AZ, USA: ACM, pp. 101–113. DOI: 10.1145/1375581.1375595. [6, 41]

Darte, Alain, Robert Schreiber, and Gilles Villard (2005). "Lattice-Based Memory Allocation". In: *IEEE Trans. Comput.* 54.10, pp. 1242–1257. DOI: 10.1109/TC.2005.167. [41]

Feautrier, Paul (Oct. 1992a). "Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time". In: *International Journal of Parallel Programming* 21.5, pp. 313–348. DOI: 10.1007/BF01407835. [6, 34]

Feautrier, Paul (Dec. 1992b). "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time". In: *International Journal of Parallel Programming* 21.6, pp. 389–420. DOI: 10.1007/BF01379404. [30]

Gannon, Dennis, William Jalby, and Kyle Gallivan (1988). "Strategies for cache and local memory management by global program optimizations". In: *Journal of Parallel and Distributed Computing* 5.5, pp. 587–616. DOI: 10.1016/0743-7315(88)90014-7. [3]

Grosser, Tobias, J. Ramanujam, Louis-Noël Pouchet, P. Sadayappan, and Sebastian Pop (2015). "Optimistic Delinearization of Parametrically Sized Arrays". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, pp. 351–360. DOI: 10.1145/2751205.2751248. [3]

Kandemir, Mahmut T., J. Ramanujam, and Alok N. Choudhary (1997). "A Compiler Algorithm for Optimizing Locality in Loop Nests". In: *Proceedings of the 11th International Conference on Supercomputing*. ICS '97. Vienna, Austria: ACM, pp. 269–276. DOI: 10.1145/263580.263650. [4]

Kandemir, Mahmut T., J. Ramanujam, and Alok N. Choudhary (Feb. 1999). "Improving Cache Locality by a Combination of Loop and Data Transformation". In: *IEEE Transactions on Computers* 48.2, pp. 159–167. DOI: 10.1109/12.752657. [3–5, 7, 12, 21]

Kandemir, Mahmut T., J. Ramanujam, Alok N. Choudhary, and Prithviraj Banerjee (Dec. 2001). "A Layout-Conscious Iteration Space Transformation Technique". In: *IEEE Transactions on Computers* 50.12, pp. 1321–1335. DOI: 10.1109/TC.2001.970571. [5]

Kong, Martin (July 2017). *Personal communication*. [9]

Kong, Martin, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan (2013). "When polyhedral transformations meet SIMD code generation". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. PLDI '13. Seattle, Washington, USA: ACM, pp. 127–138. DOI: 10.1145/2491956.2462187. [3, 8, 9]

Li, Wei (Aug. 1993). "Compiling for NUMA Parallel Machines". PhD thesis. Dept. of Computer Science, Cornell Univ. [4, 5]

Lim, Amy W. and Monica S. Lam (May 1998). "Maximizing Parallelism and Minimizing Synchronization with Affine Partitions". In: *Parallel Comput.* 24.3-4, pp. 445–475. DOI: 10.1016/S0167-8191(98)00021-0. [5]

Lim, Amy W., Shih-Wei Liao, and Monica S. Lam (June 2001). "Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning". In: *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. PPoPP '01. Snowbird, Utah, USA: ACM, pp. 103–112. DOI: 10.1145/379539.379586. [5]

Pouchet, Louis-Noël, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache (Jan. 2011). "Loop Transformations: Convexity, Pruning and Optimization". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, TX, pp. 549–562. DOI: 10.1145/1926385.1926449. [8, 9]

Schrijver, Alexander (1986). *Theory of Linear and Integer Programming*. John Wiley & Sons. [34, 44]

Trifunovic, Konrad, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen (2009). "Polyhedral-model guided loop-nest auto-vectorization". In: *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, pp. 327–337. DOI: 10.1109/PACT.2009.18. [7]

Vasilache, Nicolas (Sept. 2007). "Scalable Program Optimization Techniques in the Polyhedral Model". PhD thesis. Université Paris Sud XI, Orsay. [8, 9]

Vasilache, Nicolas (Aug. 2017). *Personal communication*. [8]

Vasilache, Nicolas, Benoît Meister, Muthu Baskaran, and Richard Lethin (Jan. 2012). "Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization". In: *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques*. Paris, France. [7, 9, 27]

Verdoolaege, Sven (2010). "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

Verdoolaege, Sven (2016). *Presburger Formulas and Polyhedral Compilation*. DOI: 10.13140/RG.2.1.1174.6323. [9]

Verdoolaege, Sven and Tobias Grosser (Jan. 2012). "Polyhedral Extraction Tool". In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France. DOI: 10.13140/RG.2.1.4213.4562.

Verdoolaege, Sven, Serge Guelton, Tobias Grosser, and Albert Cohen (Jan. 2014). "Schedule Trees". In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria. DOI: 10.13140/RG.2.1.4475.6001. [10]

Verdoolaege, Sven and Gerda Janssens (June 2017). *Scheduling for PPCG*. Report CW 706. Leuven, Belgium: Department of Computer Science, KU Leuven. DOI: 10.13140/RG.2.2.28998.68169. [6, 10, 16, 17, 24, 27, 30, 32, 34, 35, 46]

Verdoolaege, Sven, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor (2013). "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4, p. 54. DOI: 10.1145/2400682.2400713. [10]

Wolf, Michael E. and Monica S. Lam (June 1991a). "A Data Locality Optimizing Algorithm". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, pp. 30–44. DOI: 10.1145/113445.113449. [3, 5, 6]

Wolf, Michael E. and Monica S. Lam (Oct. 1991b). "A Loop Transformation Theory and an Algorithm to Maximize Parallelism". In: *IEEE Transactions on Parallel and Distributed Systems* 2.4, pp. 452–471. DOI: 10.1109/71.97902. [3]

Wolfe, Michael Joseph (1996). *High Performance Compilers for Parallel Computing*. Redwood City, CA: Addison Wesley. [4, 5]

Xilinx (Mar. 2017). Available from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1207-sdaccel-optimization-guide.pdf. UG1207 (v2016.4). [2]

Zinenko, Oleksandr, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen (Nov. 2017). *Unified Polyhedral Modeling of Temporal and Spatial Locality*. Research Report RR-9110. Inria Paris. [6, 10, 11, 46]

Index